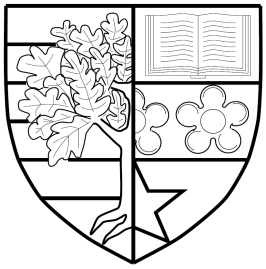
ADAPTIVE ARCHITECTURE-TRANSPARENT POLICY CONTROL IN A DISTRIBUTED GRAPH REDUCER

by

Evgenij Belikov



Submitted for the degree of
Doctor of Philosophy

Department of Computer Science
School of Mathematical and Computer Sciences
Heriot-Watt University

May 2019

Supervisors

Prof Dr Hans-Wolfgang Loidl, Heriot-Watt University

Prof Dr Greg Michaelson, Heriot-Watt University

Examination Committee

Prof Dr Henrik Nilsson, University of Nottingham

Prof Dr Sven-Bodo Scholz, Heriot-Watt University

Dedicated to my parents

Abstract

The end of the frequency scaling era occured around 2005 as the clock frequency has stalled for commodity architectures. Thus performance improvements that could in the past be expected with each new hardware generation needed to originate elsewhere. Almost all computer architectures exhibit substantial and growing levels of parallelism, exploiting which became one of the key sources of performance and scalability improvements. Alas, parallel programming proved much more difficult than sequential, due to the need to specify coordination and parallelism management aspects. Whilst low-level languages place the burden on the programmers reducing productivity and portability, semi-implicit approaches delegate the responsibility to sophisticated compilers and run-time systems.

This thesis presents a study of adaptive load distribution based on work stealing using history and ancestry information in a distributed graph reducer for a non-strict functional language. The results contribute to the exploration of more flexible run-time-system-level parallelism control implementing a semi-explicit model of parallelism, which offers productivity and high level of abstraction by delegating the responsibility of coordination to the run-time system.

After characterising a set of parallel functional applications, we study the use of historical information to adapt the choice of the victim to steal from in a work stealing scheduler. We observe substantially lower numbers of messages for data-parallel and nested applications. However, this heuristic fails in cases where past application behaviour is not resembling future behaviour, for instance for Divide-&-Conquer applications with a large number of very fine-grained threads and generators of parallelism that move dynamically across processing elements. This mechanism is not specific to the language and the run-time system, and applies to other work stealing schedulers.

Next, we focus on the other key work stealing decision of which sparks that represent potential parallelism to donate, investigating the effect of Spark Colocation on the performance of five Divide-&-Conquer programs run on a cluster of up to 256 PEs. When using Spark Colocation, the distributed graph reducer shares *related* work resulting in a higher degree of both potential and actual parallelism, and more fine-grained and less variable thread size. We validate this behaviour by observing a reduction in average fetch times, but increased amounts of `FETCH` messages and of inter-PE pointers for colocation, which nevertheless results in improved load balance for three of the five benchmark programs. The results show high speedups and speedup improvements for Spark Colocation for the three more regular and nested applications and performance degradation for two programs: one that is excessively fine-grained and one exhibiting limited scalability. Overall, Spark Colocation appears most beneficial for higher numbers of PEs, where improved load balance and higher degree of parallelism have more opportunities to pay off.

In more general terms, we show that a run-time system can beneficially use historical information on past stealing successes that is gathered dynamically and used within the same run and the ancestry information dynamically reconstructed at run time using annotations. Moreover, the results support the view that different heuristics are beneficial for applications using different parallelism patterns, underlining the advantages of a flexible architecture-transparent approach.

Acknowledgments

I am overwhelmed thinking about all the people who enriched my life, helped and influenced me in many different ways. Alas I am able to only mention a few.

I am grateful to Hans-Wolfgang and Greg for offering me the opportunity to undertake this research project and for their patience, guidance and crucial technical advice as well as detailed comments on countless drafts. Thanks to Henrik and Bodo for taking on the examination and for helpful comments that have significantly improved the thesis. Any remaining errors and poor prose are my own.

Thanks to Kons, Prabh, Akis, who not only shared our office, but also keen interest in parallelism and their friendship. Organising the AiPL'14 Summer School was fun, but also a lot of work. During my time at HWU I was fortunate to meet Hans-Nikolai, Max, Stuart, Kevin, Franta – thanks for many pub outings and board game nights. I am also grateful to friends and colleagues at MACS who provided a stimulating environment, in particular, Rob, Manuel, Phil, and Artem. Thanks to Daniel, Tim, Stuart, Gavin and Matt who taught me lots and contributed to the enjoyment of my visit at Oracle Labs in Cambridge. Thanks to David for unique leadership style and for Andy for great joint work on some among the various Data Lab projects. I am grateful to SICSA for the initial funding. Please check out and consider contributing to The Carpentries initiative that aims at teaching basic computational skills and best practices to researchers.

Thanks to all the different interest groups that offer great activities for balance: HWUMC – for epic days out in the mountains, my Jiu Jitsu family for giving hugs and throwing me into the air, Michael and Andy who share with me Systema, something you have to try for yourself, Rhubaba Gallery and Studios – for offering a refreshing programme of events over the years – extra shout out to the Situationist Three-Sided Football League. Friends from Berlin – Julian, Linda, Jonas, Markus, Jenny, Chris, among many others, I miss you all! Thanks to Siân for love and companionship over those years, and for help with the figures in Chapter 3. I am also immensely grateful to my family for continuous unconditional love and support!

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Up to around the year 2005 new generations of commodity processors have delivered immediate performance gains through increased clock frequencies. Since then, hardware performance improvements are achieved through increasing the number of cores per processor. Exploiting these cores requires parallel program execution.

Alas, parallel programming is in general more difficult than sequential, because of the added complexity of specifying coordination and parallelism management aspects, such as partitioning, mapping, communication, thread management, granularity control and load balancing. Whilst low-level languages place the burden of parallelism control on the programmers thereby reducing productivity, systems for automated parallelism delegate the responsibility to sophisticated compilers and run-time systems.

This thesis tackles the challenge of adaptive semi-explicit parallelism control, where most coordination aspects are handled by the run-time system (RTS) for distributed execution of a high-level parallel functional language with non-strict evaluation: Glasgow parallel Haskell (GpH). This RTS implements distributed graph reduction using a virtual shared heap abstraction across a distributed memory architecture, whilst also supporting shared-memory architecture.

This thesis contributes to the state of the art in load distribution by developing and evaluating novel RTS policies and mechanisms that use system-level information in a distributed graph reducer to improve application performance. In particular, we focus on historical information regarding successes of work requests and on infor-

mation about ancestry representing the relationship between tasks and sub-tasks.

The results demonstrate the benefits of more flexible architecture-transparent system-level parallelism control for a class of run-time systems implementing high-level parallel programming models. These models offer productivity and high level of abstraction by delegating the responsibility of coordination to the RTS, whilst ensuring scalability beyond a single machine by supporting distributed evaluation, with focus on a commodity cluster platform using a relatively fast non-specialised local network.

In parallel programming there is an inherent tension between performance, productivity and portability that involves a trade-off. Although fully implicit parallelism may be attainable in a restricted setting and in the longer run, whilst there always be demand for explicit programming to tune for maximum performance, we argue the tension should be resolved giving more weight to programmer productivity, whilst maintaining portable performance as much as possible in many cases. Thus, from the language design angle, we advocate the use of a high-level functional language, which offers large amounts of fine-grained parallelism. Whilst from the systems design point of view, we follow an adaptive and architecture-transparent approach, which provides the necessary flexibility. At the same time we believe in the value of diversity of approaches in helping discover improved solutions and therefore explore a less well-researched direction.

1.1 Parallelism, Programmer Productivity and Performance Portability

In the information age, our globalised society critically relies on increasingly complex and computationally demanding and distributed systems comprising diverse software and hardware components. Effectively and efficiently exploiting modern parallel architectures is key for improving application performance and scalability, for instance in areas such as Large-Scale Data Analytics and High-Performance Computing [12, 208]. This is due to a halt of the increase in CPU clock frequencies

in commodity architectures [229], because of fundamental physical limitations, such as heat dissipation, minituarisation and current leakage [165]. Since then hardware vendors managed to keep up with Gordon Moore's prediction [184] that the number of transistors would double roughly every two years by packaging more cores onto the chips. However, this no longer translates into automatic performance gains as the software often fails to fully exploit the available resources.

Mainstream parallel languages focus on explicit coordination, which allows maximum control over synchronisation between tasks and data transfer, but also imposes the responsibility of specifying coordination on the programmer, thus reducing productivity. Even worse, these programs are prone to a new class of errors that are notoriously difficult to detect and correct, such as race conditions and deadlocks [149], whilst the results of stateful computations may become non-deterministic. Thus when programming in such relatively low-level languages, program development tends to be more time-consuming due to the need to explicitly encode synchronisation and communication, whilst ensuring correctness. Therefore there is a need for parallel programming models with higher levels of abstraction to tame the complexity and allow more flexibility to adapt to changing architectures and execution environment to maintain high performance.

Exploiting parallelism is inherently more difficult as modern computer architectures are increasingly *diverse*: hierarchical with non-uniform memory access (NUMA), e.g. as in multi-socket machines, and heterogeneous with qualitatively different processing elements (PEs[1]) including CPUs, GPUs, co-processors as well as DSPs and FPGAs [190, 43]. Most of these architectures have different trade-offs, bottlenecks and programming models for specifying the coordination aspects of parallel execution such as identification of parallelism, partitioning and aggregation to specify suitable granularity, work distribution, communication and synchronisation, underlining the need for adaptivity. In this thesis we focus on *distributed-memory architectures* in the form of *commodity clusters*, which offer an affordable and scalable high-performance platform. Cloud Computing and Heterogeneous Computing, though popular, remain out of the scope of this work, posing separate challenges.

[1]here we also use PE as a synonym for an RTS instance that runs on the physical PE

1.2 Semi-Explicit Parallel Functional Programming

Functional programming is often cited as beneficial for programmer productivity due to the *declarative style*, where the programmer is focused on problem specification rather than on describing how the problem is to be solved as a sequence of instructions [16, 244, 252]. The *high level of abstraction* is supported by key language features such as *higher-order functions*, *polymorphism*, and *type classes* [84, 126, 121].

In particular due to the Church-Rosser property [63], functional languages appear suitable for exploitation of *fine-grained parallelism* as independent sub-expressions can be evaluated in any order, especially in parallel, without changing the resulting value. Although some care is required to avoid changing the program's termination behaviour by ensuring that parallel expressions are shared with the remainder of the computation and correct evaluation order is preserved. Moreover *referential transparency* [227] and isolation of side-effects, enables the programming model that is *deterministic* by design, guaranteeing the absence of race conditions and deadlocks, a class of errors that is very difficult to diagnose and eliminate [149]. However implementing such a language may still require the use of some unsafe mechanisms at the systems level shifting the difficulty away from application-level programmers.

This facilitates *incremental parallelisation*, where sub-expressions can be gradually marked for parallel evaluation without affecting the final result, and allows for *sequential debugging* of parallel programs, where the program can be debugged using existing sequential debugging tools with parallelism turned-off and the result of parallel execution is deterministically[2] guaranteed to be the same [109].

Similarly, a high-level mostly-implicit approach to parallelism can provide a unified architecture-independent programming model where the programmer is focused on specifying the computation, whereas the coordination is mostly managed by the compiler and the RTS. Compared to explicit approaches, it increases system-level adaptive flexibility for tuning to diverse target architectures and applications.

Moreover, additional flexibility can be obtained through the use of *advisory* rather than *mandatory* RTS policies offering opportunities for adaptive tuning. This

[2] however the performance may differ substantially, requiring parallel profiling for tuning

distinction is not specific to functional programming languages, many implementations of which actually use mandatory policies. The latter policies are inflexible due to forcing a particular decision, whereas the former leave the final decision to the compiler and the RTS, which may utilise dynamic system information to make better decisions depending on the given situation at run time, whilst potentially disregarding any hints provided by the programmer.

Further advantage of the functional approach to parallel programming results from the use of HOFs to capture common parallelism patterns in a composable way facilitating the implementation of algorithmic skeletons [66, 206, 103].

As most of the responsibilities are shifted to the RTS, the challenge is thus the efficient implementation of adaptive parallelism control inside a language RTS.

1.3 Adaptive Architecture-Transparent Control of Parallelism

Due to high complexity of both software and hardware architectures, explicitly encoding thread synchronisation and data exchange patterns as well as thread placement is deemed inhibitively difficult in many cases, as it becomes more time and resource consuming as well as less likely to be close to the optimal solution, e.g. minimal run time, within the rapidly growing multi-dimensional optimisation space.

A relevant observation is that in many cases the software tends to outlive the hardware it initially used to be run on. Often there is a significant difference between target architectures of different generations and vendors. For example many HPC platforms used flat clusters in the past, whilst currently most HPC clusters are hierarchical due to use of NUMA nodes. Thus changing the underlying platform requires re-tuning, which is a tedious and ad hoc process based on deeply specialist knowledge. This may result in less portable code and performance degradation once the assumptions made in the past no longer hold.

To achieve high programmer productivity, architectural details should be hidden, as much as possible, within a compiler and the run-time system, whilst allowing some

architecture awareness at those lower levels to enable adaptation to architectural and system-level changes. Additonal flexibility results in additional overhead that should be kept minimal.

Our Approach: We tackle the challenge of efficient semi-explicit parallel programming at the RTS level, because at this level most relevant information is available and can directly influence the evaluation of the running application. Architecture-transparency and adaptivity offer the needed flexibility of control of parallelism without sacrificing performance, whilst an expressive high-level language caters for the productivity needs.

Moreover, *advisory parallelism* is used for additional flexibility in granularity control to better adapt the degree of parallelism to the given static number of PEs and dynamic application characteristics. Potential parallelism is represented in the RTS, allowing it to be either turned into actual parallelism in the form of light-weight threads if idle PEs are available, or sequentially executed otherwise.

We use a random work-stealing mechanism for load distribution as the baseline for comparison [38, 237]. It has grown in popularity over the last two decades because of its scalability, which is due to its decentralised core algorithm that requires no global knowledge. However, due to randomisation the operational behaviour is non-deterministic, even though the delivered results are deterministic, which complicates the analysis and requires in-depth system profiling.

We develop and evaluate adaptive heuristics and mechanisms that can respond to changing conditions within the same application run. In particular, load balance can be improved without directly monitoring the load. Adaptivity enables the RTS to react to phase changes within the application, to input-dependent differences that are difficult to anticipate in irregular applications, and to variations in parallelism degree during the execution by utilising historical system-level information, as well as information on the ancestry of parallelism recovered during compilation and dynamically reconstructed by the RTS. Historical information can help de-randomise work-stealing behaviour when parallelism generation patterns are spatiotemporally stable, whilst co-locating related parallelism is aimed at improving load balance.

1.4 Contributions and Authorship

This thesis makes the following contributions:

1. The design, implementation and empirical evaluation of a Spark Colocation mechanism for allocating advisory parallelism to improve application performance through better load balance and locality. The mechanism uses ancestry information which encodes the location of potentially parallel computations (sparks) in the computational graph making it available to the RTS, which uses this information by selecting related sparks for donation in response to work requests based on maximum prefix matching on the encodings.

2. The design, implementation and empirical evaluation of a History-Based Work Stealing mechanism, which uses partial monitored information on past stealing successes and failures within the same application run to select PEs to request work from.

3. The characterisation of several small- to medium-sized applications written in a non-strict parallel functional language using end- as well as means-based metrics on a multi-core and a cluster of multi-cores. This demonstates scalability limitations of aggressive load distribution and provides the rationale for the introduced mechanisms.

4. The comprehensive survey of support for distributed graph reduction and an overview of language run-time systems with support for adaptive parallelism control, as well as the introduction of a classification scheme for RTS mechanisms for adaptive management of parallelism.

The work presented in this thesis is based upon and extends several publications and technical reports. Chapter 2 includes information from a technical report summarising high-level parallel programming models [24] (TR #1 in the list below) and from an overview paper covering language run-time systems [23] (Paper #3). The material reported in Chapter 4 has been published [25] (Paper #2), whilst results presented in Chapter 5 extend an earlier publication [21] (Paper #1). A paper based

on the results from Chapter 6 has been accepted for publication [26] (Paper #4). Some of the implementation details overlap with information provided in a companion technical report [22] (TR #2). All of the publications are available online as Open Access documents. The full list is provided below.

Publications

1. E. Belikov. History-based adaptive work distribution. In *Proc. of Imperial College Computing Student Workshop*, volume 43 of *OpenAccess Series in Informatics (OASIcs)*, pages 3–10. Leibniz-Zentrum fuer Informatik, 2014

2. E. Belikov, H.-W. Loidl, and G. Michaelson. Towards a characterisation of parallel functional applications. In *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering, Dresden, Germany*, pages 146–153, 2015; as the first author, the author co-designed the study, run the experiments, analysed and evaluated the data, and wrote the bulk of the paper.

3. E. Belikov. Language Run-time Systems: an Overview. In *Proc. of Imperial College Computing Student Workshop*, volume 49 of *OpenAccess Series in Informatics (OASIcs)*, pages 3–12. Leibniz-Zentrum fuer Informatik, 2015

4. E. Belikov, H.-W. Loidl, and G. Michaelson. Colocation of potential parallelism in a distributed adaptive run-time system for parallel Haskell. In *Proceedings of the International Symposium on Trends in Functional Programming, Gothenburg, Sweden*, pages 1–19. Springer, 2018; as the first author, the author co-developed the idea, run the experiments, analysed and evaluated the data, and wrote the bulk of the paper.

Technical Reports

1. E. Belikov, P. Deligiannis, P. Totoo, M. Aljabri, and H.-W. Loidl. A survey of high-level parallel programming models. Technical Report HW-MACS-TR-0103, Dept of Computer Science, Heriot-Watt University, Dec. 2013; as the first author, the author initiated the effort, acted as lead author for sections 1, 2, 3, in particular 6 (on run-time systems), 8, and edited the whole paper.

2. E. Belikov. Hitchhiker's guide to GUM hacking. Technical Report HW-MACS-TR-0112, Dept of Computer Science, Heriot-Watt University, Dec. 2015

1.5 Outline

The structure of the rest of the thesis is as follows:

- Chapter 2 provides the context based on literature from the areas of parallel architectures, parallel programming models, parallel functional programming, parallel run-time systems and adaptive parallelism management.
- Chapter 3 describes distributed graph reduction in the Graph reduction on a Unified machine Model (GUM) RTS that is used to execute Glasgow parallel Haskell programs, its abstract model, design decisions, and implementation details related to key policies and mechanisms. In particular, the focus is on load balancing and work stealing. Additionally, an adaptivity classification scheme is introduced.
- Chapter 4 investigates the characteristics of parallel functional applications and, building on Chapter 3, gives the rationale for extensions for the default load balancing mechanism.
- Chapter 5 introduces and evaluates History-Based Stealing where the choice of PEs to steal from is no longer completely random, but biased based on the historical information on past stealing successes to improve future stealing success rate.
- Chapter 6 discusses the design, implementation and evaluation of Spark Colocation, that is colocation of potentially parallel work (sparks) using the knowledge about the location of the spark within the computation graph at run time to decide which spark to donate, given the ancestry of the requesting thread.
- Chapter 7 concludes, discusses the limitations of this work and suggests several directions for future research.

Chapter 2

Background

This chapter presents relevant background information on concepts related to adaptive architecture-transparent control of semi-implicit parallelism including current and historical developments, using information from a co-authored technical report [24], among other literature.

We focus on the following key areas. We begin with a discussion of parallel architectures in Section 2.1, including Flynn's taxonomy, Skillicorn's taxonomy, shared-memory as well as distributed-memory architectures, uniformity, hierarchy and heterogeneity, and summarise the perceived consequences of the recent architectural trends for language design and implementation.

Section 2.2 describes different high-level approaches to exploiting parallelism with a focus on parallel functional programming languages, classifies approaches to coordination based on the level of implicitness, and contrasts different languages and libraries. We also discuss key concepts such as strictness and laziness.

In Section 2.3, we discuss practical implementation challenges associated with the non-strict functional approach and abstract-machine-based compiled graph reduction in a distributed setting. A detailed overview of the GUM RTS for Haskell is provided in Chapter 3.

Finally in Section 2.4, we focus on adaptivity in run-time systems, covering coordination policies and mechanisms such as scheduling, load balancing, memory management, communication, and granularity control.

2.1 Parallel Architectures

This section summarises the rapidly changing landscape of parallel architectures. In particular, we focus on distributed-memory architectures which offer high levels of scalability beyond a single node [208]. Detailed discussion of GPUs [190] and FPGAs [143] is outside of the scope of this thesis.

Computer architecture describes the organisation and the capabilities of the processing elements (PEs), memory units, controllers and the interconnection network. The first digital computers had only a single central processing unit (CPU) that could process a single stream of data, where code and data shared the same memory. This is also called the *von Neumann* architecture to honour one of the major pioneers of Computing Science [247]. Subsequently, different designs were explored, and Flynn established a taxonomy to classify computer architectures based on the number of instruction and data streams [87, 88, 89], single or multiple, resulting in four classes: SISD, SIMD, MISD, and MIMD.

Although criticised for its bottleneck between the processor and memory that limits system performance [16, 258], von Neumann or SISD architecture was popular for a long time until frequency scaling became economically unprofitable around 2005 due to fundamental physical limitations [229]. In particular continued miniaturisation and wire fan-out, needed to further increase the number of transistors, resulted in higher current leakage, and heat dissipation issues, because of increased power needs, also referred to as power wall [40, 165]. The effect was mitigated to some extent through the use of gradually more sophisticated pipelining, pre-fetching and caching mechanisms and out-of-order execution, but it couldn't be eliminated.

However, Flynn's classic taxonomy is now no longer sufficient to distinguish between different architectures, as most modern architectures fall into the same MIMD category according to Flynn. Graphics Processing Units (GPU) provide a prominent example of a SIMD architecture, and are now increasinly used for general-purpose computation [190]. The rather esoteric MISD architecture, e.g. systolic arrays [142], found use in Aerospace [53] due to increased fault tolerance. More recently, Google has released the Tensor Processing Unit (TPU) [210] designed

around a 256×256 systolic array of matrix multipliers to accelerate computations in Convolutional Neural Networks.

Flynn's Taxonomy was extended from 4 to 28 classes by Skillicorn [222] to increase its discriminatory power based on explicit assessment of coupling between processors and using state machine view of processors at finer granularity, but turned out too complex to become widespread.

In practice, the MIMD category is commonly sub-divided into *distributed-* and *shared-memory* architectures based on the accessibility of the physical memory [78]. Another distinction is made between *uniform* and *non-uniform* memory-access for the latter (UMA and NUMA repectively), distinguished by the topology and speed of interconnect as well as on computational power of the nodes in the network. Furthermore, the requirement for and implementation of *cache coherence* can be used as an additional distinguishing characteristic [225]. Finally, with increasing heterogeneity, further features can be used such as the number and type of processing units. Next we discuss the main sub-classes.

2.1.1 Shared-Memory Architectures

Shared-memory architectures provide a unified view of a single address space to all PEs and are designed so that PEs have access to all physical memory without the need to send off-node messages over a network. The advantage of such architectures is that often interconnect is orders of magnitude faster than using an off-node network connection.

From the programming model angle, a global memory view was deemed convenient as it allows for every thread to access any memory region, but gradually explicit synchronisation became recognised as detrimental for performance and programmer productivity, because of contention and locking granularity issues, as well as due to non-deterministic errors [189, 149]. Scalability is limited due to the global view of memory which leads to increasing overheads for maintaining cache coherence, i.e. a consistent view of memory across all the PEs [225].

Uniform Memory Access (UMA) architectures ensure that memory access results

in the same costs no matter from which PE and which memory region is accessed. This category constitutes a sub-category of shared-memory architectures and subsumes Symmetric Multiprocessors (SMPs). With growing numbers of PEs it proved challenging to maintain uniform access which resulted in the Non-Uniform Memory Access (NUMA) architectures.

NUMA architectures are now more common as memory controllers are placed closer to and integrated within the processors [144] and are more cost-effective in production at the price of sacrificing uniformity.

Modern server-class architectures exhibit tens of cores spread across multiple sockets with several cores sharing a socket. The access and communication cost may differ by a factor of two or more depending of whether PEs are sharing a socket and whether nearby or further-away memory bank is accessed. This non-uniformity introduces non-trivial process placement trade-offs making data locality an increasingly important concern [151]: even more so in a distributed setting where communication cost is very high compared to computation.

2.1.2 Distributed-Memory Architectures

Distributed-memory architectures avoid sharing memory resulting in a more scalable design. PEs and networked nodes have separate physical memory and communicate via message passing, which reduces coupling and avoids cache coherence issues. However, it is possible to implement software abstractions that hide communication and synchronisation on top of the architecture such as virtual shared-memory and the Partitioned-Global-Address-Space (PGAS) [254], which can provide an illusion of shared memory.

A cluster comprising multiple nodes which could contain several PEs each is a widely used distributed architecture. For instance Beowulf clusters [209] gained popularity due to their favourable cost-performance ratio and the off-the-shelf availability of software and hardware components. Similar to Networks-of-Workstations [5], clusters often use relatively slow networks such as Ethernet to establish local area and wide-area network connections. Computational Grids [93, 92, 91, 94], Rackscale

systems [68, 112], Cloud systems [8] and most Supercomputers (87.4% of Top500 list entrants in June 2018 [76]) utilise clusters of networked machines at the underlying infrastructural level, often using a fast interconnect such as Infiniband or Myrinet, which is more expensive, but offers an order of magnitude lower latency and higher bandwidth than Ethernet.

Although we run our experiments on a mainstream distributed architecture, we briefly discuss trends towards heterogeneity for completeness as such architectures are increasingly more commonly used.

2.1.3 Heterogeneous Architectures

As described above, most systems are hierarchical to some extent in terms of communication costs. Accessing files from a hard disk and communicating over an off-chip network is orders of magnitude slower than accessing working memory or on-chip cache [117], with recent improvements on both sides through Solid State Disk (SSD) and Non-Volatile Memory (NVM) technologies. Additionally, modern architectures are becoming more *heterogeneous* with respect to the number and capabilities of the avalialble PEs.

For example, chips such as IBM Cell BE found in Sony's Playstation contain both conventional cores and synergistic processing units (SPEs) that use RDMA for data transfer [57]. Similarly, ARM's big.LITTLE features both more traditional CPUs and several smaller and more power-efficient cores [132]. On the other hand companies such as Tilera and Parallela are promoting tiled architectures using homogeneous cores [207, 106].

Another example is Intel's Xeon Phi, a 60-core general-purpose co-processor supporting a typical x86 instruction set with extensions for vectorisation [59]. Despite being connected via PCIe (as most GPUs are), this allows MPI to be used as communications library. It offers higher maximum performance than comparable Xeon processors, but to fully exploit it the code needs to be vectorised [133, 59].

GPUs have recently gained attention as a power-effcient way to improve parallel performance of suitable data-parallel application. Speedups of 100x and even 1000x

over CPU performance have been reported [150, 104]. In particular, independent data-parallel computationally intensive workloads that operate on floats or doubles benefit from massively parallel execution.

Other processors are specialised for particular applications[1] such as for Network Traffic Monitoring and Digital Signal Processing (DSPs), Fast Fourier Transform or Cryptography. Depending on the applciation area, such solutions are potentially available off-the-shelf, despite being costly to build and maintain.

Some architectures, such as Field Programmable Gate Arrays (FPGAs) [143], support the possibility to reprogram the functionality and grouping of gate arrays to suit application demands and are suitable for applications that use integer-based and logic operations, with cores running at lower frequency than traditional CPUs.

Novel experimental architectures have been proposed based on the System-on-Chip design [257], including the key components on the die area thus resulting in very fast intrachip communication able to reach higher throughputs. Additionally, Dark Silicon refers to the inability to simultaneously power on all the available transistors [80], so it is conceivable that in the future architectures will to some extent have the ability to change on-the-fly based on application requirements.

2.1.4 Implications for Language Design and Implementation

As almost all architectures are now parallel, and all large-scale architectures are distributed to overcome scalability limitations of single-node solutions, there is a need for native parallelism and distribution support at both the language design and implementation levels.

Additionally, increasing architectural diversity substantially complicates manual performance tuning. From this perspective, a RTS appears well-suited to provide adaptive yet mostly transparent support for parallelism and distribution. For example, the RTS can transparently arrange communication and synchronisation without the need for the programmer to intervene, whilst adapting the number of light-weight threads based on monitored system load. Next, we describe parallel programming

[1] also called Application-Specific Integrated Circuits or ASICs

languages for distributed parallel architectures, with particular focus on non-strict functional languages.

2.2 Parallel Functional Programming

This section describes parallel functional programming, a increasingly more popular approach with mathematical foundations in the λ-*calculus* [62, 19, 18] that offers a high level of abstraction, whilst being suitable for safe and flexible exploitation of parallelism inherent in many applications. We also discuss distributed computing as a way to improve application scalability and utilise powerful large-scale parallel architectures.

Moreover, we argue that high-level languages can help battle increasing software complexity by avoiding placing the burden of explicitly specifying coordination decisions on application-level programmers. This can both increase productivity, which is critical if we are to meet the growing societal computational demands, and avoid overspecification, which we claim can be considered harmful given the substantial architectural diversity and the disadvantages of manual tuning.

First, we review definitions of concurrency and parallelism and discuss the advantages of the functional approach, as well as its fundamental properties that are beneficial for exploiting parallelism. Next, we discuss coordination aspects relevant for efficient exploitation of parallelism. Furthermore, we describe key differences between the *strict* and *non-strict* semantics as we use a non-strict language in our studies. Finally, we discuss several prominent functional programming languages and the abstractions they support, including language features, libraries, skeletons, and patterns, with a focus on expressing parallelism. The discussion of implementation and distributed execution follows in Sections 2.3, whilst adaptive parallelism management is reviewed in Section 2.4.

We exclude detailed discussion of imperative and object-oriented as well as logic parallel programming languages due to the breadth of the area and the relatively low levels of abstraction of the former languages. Surveys and historical information on these approaches can be found in the literature [261, 220, 58, 248, 103, 74].

2.2.1 Concurrency and Parallelism

Concurrency and parallelism are related concepts and different definitions may overlap, which can lead to confusion. Whereas processes running on a sequential computer need to take turns executing on the single CPU they share, using parallel hardware enables multiple processes to execute truly simultaneously on different processors or cores.

On the one hand, the broad definition of concurrency refers to several events happening without a specified ordering. Thus concurrent processes could proceed simultaneously, interleaved or sequentially. In particular, if dependencies exist some order may be imposed for concurrent processes to synchronise. This definition subsumes parallelism, if parallelism is defined as strict simultaneity.

On the other hand, in programming, concurrency is often defined more narrowly with reference to the execution of threads and as a program-structuring technique to modularise programs that need to handle multiple tasks. The goal is both modularisation and performance by providing responsiveness and asynchronous processing, e.g. when handling I/O and processing in a GUI, or responding to multiple requests in a web server.

Moreover, parallelism usually implies the use of parallel architectures. Thus we define parallelism as simultaneous execution of multiple processes or threads on physical parallel hardware. The goal of parallelism is performance improvement, which can manifest itself as decreased runtime, higher output accuracy, or through the ability to handle larger inputs. The definitions we are using are inspired by those recently suggested by Marlow [166].

2.2.2 Why Parallel, Why Functional

As most computer architectures are now parallel and many algorithms exhibit some degree of inherent parallelism, whilst ever larger amounts of data need to be processed, the need to exploit parallelism becomes apparent [11]. Otherwise, if potentially parallel algorithms are expressed sequentially, much potential for performance improvement is wasted and the opportunity to handle larger inputs with respect to

data as well as computation is missed.

In an influential paper [126], Hughes emphasised the suitability of functional programming as a new kind of "glue" providing new high level features, in particular higher order functions (HOFs) and lazy evaluation. HOFs are functions that may take other functions as arguments and return functions as results. Lazy evaluation is also called *call-by-need*, where an expression's evaluation is delayed until the result is demanded by another computation. These features facilitate structuring of complex programs, modularisation, reuse and problem decomposition as well as separation of concerns, which is more difficult to achieve in languages with unrestricted side-effects. Using HOFs and laziness, programs can be concisely expressed as a composition of functions working on potentially infinite data structures, as also advocated by Turner [244]. Furthermore, a powerful type system helps ensure correctness at compile time.

The functional approach also appears suitable to exploit high-level parallelism in a structured way by offering higher levels of abstraction resulting in benefits for programmer productivity, as Hammond eloquently argued in a panel statement [109]. Additionally, a flexible execution environment naturally complements a high-level language. In particular, using pure functional languages that avoid and isolate side-effects enables parallel execution of sub-computations, facilitating partitioning and shifting the challenge to granularity control and load balancing, whilst preventing deadlocks and race conditions. Moreover, the application can be developed and debugged sequentially and could then run in parallel and deterministically deliver the same result, whilst taking full advantage of the parallel hardware. This allows for incremental profiling-based parallelisation, rather than requiring the programmer to completely rewrite the program.

Next we review the key concepts associated with the functional approach [122].

2.2.3 Fundamental Concepts

Functional programming is mathematically founded on the λ-*calculus* [19, 18], invented by Church in 1930s to describe effectively computable functions [61, 62],

and can be considered the first universal general-purpose programming language, despite its appearance well before the first digital computers became operational. It has been shown to be equivalent to other prominent ways to describe computable functions – the Turing Machine [242, 241] and general recursive functions [138].

In λ-calculus, a term is an expression that is a variable that refers to a value or a function, a function application, or an abstraction which results in binding of free variables. Thus, a program is an expression to be evaluated by applying the conversion and reduction rules in the context of function definitions. The power of the λ-calculus arises from the ability of functions to be applied to themselves. There exist different flavours of the calculi, both untyped and typed.

A key relationship with respect to evaluation order is described by the *Church-Rosser* theorem stating that any valid order of reductions will lead to the same *normal form* value (where no further reduction is possible), starting from the same original expression [63], if reduction sequence terminates [19][2]. Care must be taken to ensure termination behaviour is not affected through the chosen evaluation order, e.g. through the use speculative parallelism that can lead to a more strict evaluation degree than in the sequential non-strict program. In general this facilitates equational reasoning and is beneficial for exploiting parallelism, if it is possible to decide which redexes can be evaluated without causing infinite computation [46]. The interested reader may refer to Hudak's survey [122] and Turner's account [246] for a history of functional programming.

Purity is a key concept, which refers to functions with no side-effects that can be treated like mathematical functions. Imperative languages are focused on mutating state and allow implicit state changes to potentially global variables at any point in the program. However in purely functional languages, there is no assignment operation and with it a whole class of errors is eliminated, making the order of execution irrelevant for correctness, and avoiding the need for the programmer to explicitly prescribe the flow of control. Thus such languages exhibit *referential transparency* [227] and variables can be replaced by the values they denote, enabling the use of *equational reasoning* to prove the correctness of optimising transformations.

[2] in other words it shows that λ-calculus is confluent under β-reductions

Another key concept is *currying*[3], which allows uniform treatment of all functions as nested function applications of functions of only one argument without loss of generality. This is possible by expressing a function of arity n as a function applied to the first argument returning a function of arity $n-1$. This expects the remaining arguments, of which it may be partially applied to the first one to return a function of arity $n-2$ and so forth until all arguments are consumed, e.g. $f(a,b) = (fa)b$.

An important branch of research associated with functional programming is *type theory* and practical implementation of type inference, in particular the notable Hindley-Milner [120, 180] type system developed for ML as a restricted parametric polymorphic type system for which type inference is decidable. Static strong typing discipline has been shown to be beneficial by avoiding whole classes of type errors at compile-time, thus saving debugging effort and cost associated with finding such errors at run time and in production. The related *Curry-Howard* isomorphism suggests that types can act as propositions and programs as proofs. Additionally, many deep connections exist to the *Category Theory* branch of mathematics.

Moreover, imperative languages such as C++, C#, Java, JavaScript, and Python now include lambdas (anonymous functions) as part of their features, with pattern matching proving more difficult to implement given the semantics of the respective language.

We discuss strict and non-strict sematics in Subsection 2.2.5. What follows is the description of the coordination aspects of exploiting parallelism.

2.2.4 Coordination Aspects

The added complexity of parallel programming stems from the need to arrange coordination, including access to shared resources, synchronisation and inter-process communication. This in addition to computation, which needs to be decomposed and data to be partitioned and mapped to threads which are in turn mapped to processors or cores, to enable parallel execution. For example, the popular PCAM parallelisation methodology by Foster [90], which is independent of the program-

[3]named after Haskell B. Curry, a prominent logician after whom the language Haskell is named

ming language used, stands for partitioning, communication, agglomeration, and mapping. These all refer to coordination aspects defining how a task is split into sub-tasks, which dependencies are present that require communication, and how tasks can be re-combined to increase granularity and mapped to PEs to minimise inter-PE communication. Most of the coordination decisions are NP-hard and some of them reflect direct trade-offs, such as the one between communication and re-computation. Different approaches to parallelism can be characterised based on the level of explicitness [223] in expressing coordination aspects, i.e. whether it is the programmer's or the system's responsibility to deal with coordination aspects.

Although they allow for maximum control, low-level programming models, such as C+MPI, are often fully *explicit* and require the programmer to specify all coordination aspects in addition to providing a correct solution to the actual problem in a specific domain. By contrast, higher-level languages, such as parallel functional programming and skeleton-based approaches, are often *semi-implicit* by hiding some of the coordination aspects from the programmer and thus increasing productivity and allowing the run-time system to flexibly adapt application behaviour to the changing execution environment and application demands.

Low-level approaches are often *prescriptive*, requiring mandatory thread creation when a particular language construct or function for thread creation is used. This can lead to overspecification of the parallelism degree and evaluation order. One exampleis manually setting the number for MPI processes. This requires re-tuning of the application if it is moved to a new target platform with a different number of PEs or when application parameters are changed, as the original assumptions may no longer hold. Rapid hardware evolution adds further difficulty to manual tuning in many cases, emphasising the need for more flexible, automated and adaptive solutions. Here we focus on the high-level approaches. On the other hand, the *advisory* approach to parallelism management, where the user provides non-binding hints to the RTS, results in additional flexibility by leaving the final decision with the system, helping to avoid overspecification.

Thus achieving efficient parallel execution is challenging, as all coordination

mechanisms contribute to the parallel overhead [44], which may end up cancelling out the benefits from exploiting parallelism. Using a high-level approach, this complexity is generally pushed from application-level programmers to system-level programmers and from application-level code to language implementations and libraries that can be reused and so application code can benefit from future updates. However, we acknowledge that in many cases static and explicitly tuned solutions reach the highest levels of performance, which may justify higher development and maintenance costs.

From the point of view of reusability and flexible adaptability, fully implicit approaches are appealing – sequential code can run in parallel without any extra effort from the application programmer. However, some degree of explicit tuning may also be desirable and so far it proved very difficult to achieve scalable implicit parallelism in practice in a general setting, due to the complexity of the coordination decisions. Most successful examples are limited to restricted forms of parallelism such as algorithmic skeletons [66, 206, 103], which encapsulate common parallelism patterns, or to cases where there are no dependencies. Prominent examples are Google's MapReduce [72] for the former, and Monte Carlo simulation [56] for the latter, and cases that are restricted to certain data structures such as streams or arrays [178, 214].

Another approach uses a functional intermediate language with rewrite rules to generate viable versions of a given program and select the best version by profiling, reporting performance comparable to hand-optimised versions in OpenCL [226].

In the non-strict high-level setting, recent examples of implicit parallelism are Feedback-Directed Implicit Parallelism (FDIP) [114], where sequential Haskell programs are profiled to identify parallelism that can be used to reach speedups of 10-80% for 7 out of 20 'nofib' benchmark [192] programs on a 4-core shared-memory machine. In Calderon Trilla's thesis [50], run-time-directed and profile-directed iterative feedback are used to improve over static analysis in utilising parallelism, reporting speedups of 3.31× and 1.64× for two out of six benchmark programs when translated to Haskell from a custom intermediate language[4]. FDIP authors

[4]simulations showed speedups of up to 15× (on 16 cores) depending on the number of cores

emphasise that their approach is not a replacement, but rather complimentary to existing semi-explicit programming models. These results show that it is possible to exploit modest amounts parallelism "for free", but also that this amount is application and input dependent and thus more research is needed to be able to detect applications that are likely to benefit from fully implicit approach at all.

As at present we cannot realistically rely on fully implicit parallelism in general, it appears that a limited number of annotations by the programmer are helpful and we can come close to the ideal in the semi-explicit case, where identification of parallelism is explicit whilst other coordination aspects remain implicit, with the exception of crude application-level granularity control through thresholding or through more sophisticated fuel-based[5] techniques [231].

Types of Parallelism Broadly speaking parallelism can be sub-divided based on a focus on either control or data [191]. *Task parallelism* emphasises the computation and how it can be split into sub-tasks that can be executed in parallel. Any dependencies and ordering requirements have to be enforced through synchronisation. By contrast, *data parallelism* is focused on the data and how data can be decomposed into chunks for multiple PEs to work on multiple data units simultaneously. As opposed to operations on scalars, vector operations are designed to perform the operation on each element of the vector in parallel. Both approaches can also be combined and nested. Moreover, *loop parallelism* can be considered as either task or data parallelism based on the loop body and parallelism exploited by executing multiple independent loop iterations simultaneously [205].

2.2.5 Strict versus Non-Strict Semantics

Denotational semantics [216] describes *what* is to be computed by formally assigning meaning to expressions written in a given programming language, whilst *how* the computation is performed is defined through the operational semantics [204] and the actual implementation.

and chosen sparking cost in the number of reductions

[5]fuel is the quantity of parallelism initially assigned to a function that is spend during parallel evaluation; once fuel reaches zero the evaluation is switched to sequential

A language is called denotationally *strict* when the expression `f` $\perp$ always evaluates to $\perp$, where $\perp$ (pronounced as bottom) refers to an undefined computation, whilst otherwise the language is called *non-strict*. Thus in a non-strict language, the expression `f` $\perp$ may refer to a value despite the argument being undefined.

An example of a non-strict language construct that is used in most languages is the conditional expression[6] `if cond ... else ...` that chooses which branch to execute depending on the Boolean `cond` *at run time*. Similarly, boolean operators supporting *short-circuit evaluation* can be considered non-strict, because some arguments may be skipped during evaluation if they do not contribute to the final value of the whole expression[7].

2.2.6 Parallel Languages and Abstractions

This subsection gives a brief overview of parallel functional languages, including history, recent developments, and key language features to express both computation and coordination at a high level of abstraction. More details on the history of functional programming are provided by Hudak [122] and Turner [246].

The potential for parallelism due to purity has been widely recognised early on, but initially much effort focused on producing an efficient sequential implementation to challenge the past sentiment that functional languages are inherently inefficient. In particular, Lisp [176, 177][8], although originally based on Kleene's theory of first-order recusive functions rather than λ-calculus (which is now the basis of modern Lisp implementations) introduced many influential concepts. Lisp is impure due to the use of assignment and `goto`, and supports S-expressions, garbage collection, and metaprogramming using `eval`, `quote` and `apply` functions.

Another influential family of languages is ISWIM [146][9] based on λ-calculus with `let`, `rec`, and `where` syntactic sugar, with offside-rule for scoping. This also included assignment and a generalised jump operator [147], designed for evaluation by an abstract machine (SECD).

[6] its invention is attributed to McCarthy during the development of Lisp in the 1950s
[7] e.g. if `a` in `a AND b` evaluates to `false`, so immediately does the whole expression
[8] Lisp stands for "LISt Processor" and was designed for symbolic and AI applications
[9] which short for "If you See What I Mean"

The reasons why initial implementations of functional languages were slow include the use of interpreters, hardware with very limited amount of memory, and use of call-by-name. This radically changed during the next several decades. In his thesis, Wadsworth proposed normal-order graph reduction that avoided work duplication through sharing and in-place updates [253], which was applied by Turner to Curry's SK-combinators [71] when implementing SASL [243], using a fixed set of combinators with some additional derived combinators for efficiency. Related work offered arguments in favour of lazy evaluation, in particular that `CONS` should not evaluate its arguments [251, 116, 96]. In the 1970s there was broad excitement about the potential of the functional approach, with Backus presenting a strong argument in his acclaimed Turing Award lecture [16]. A lot of research commenced in the 1980s on functional languages [122], abstract machines [139], as well as special-purpose architectures [250].

This strand of research was additionally motivated as a response to the ambitious Japanese research programme on 5th-Generation Computing [219] to build large parallel processing systems and corresponding programming languages with particular focus on AI. A comprehensive overview of the European ESPRIT-415 project exemplifies the breadth of explored parallel languages and architectures [234].

The introduction of supercombinators [128] and λ-lifting [134] enabled the development of compiled graph reduction [15], coinciding with a motion away from custom-build hardware, such as NORMA [212], ALICE [70] and GRIP [199] machines, towards compilation for execution on commodity hardware [193, 195][10]. Additionally, many similar experimental functional languages existed at the time, leading to a consensus that it would be useful to define a standard language for research and teaching, culminating in the development of Haskell [124, 196, 172, 123]. Haskell was inspired by Miranda [245], SML [181], and Hope [48], among other languages, most recent standard being Haskell 2010 (with upcoming Haskell 2020).

Another strand of research focused on implementation of parallel functional languages [194, 215, 108], and initially many attempts were made to provide implicit

[10]the Stackless Tagless G-Machine (STG) being the basis of current Haskell implementation in the de facto standard Glasgow/Glorious Haskell Compiler (GHC)

parallelism, whilst simultaneously defining a suitable special-purpose architecture for the language [250]. In the context of the ZAPP architecture [173], Burton and Sleep explored ways to evaluate sub-expressions on a virtual tree of processors in parallel [49]. They found it beneficial to initially generate parallelism in a breadth-first way until all the PEs were busy and then switch to depth-first strategy, which can be considered adaptive with respect to the number of PEs and the amount of parallelism in the given program.

Meanwhile, Kindgon, Burn et al., had investigated distributed execution of non-strict functional programs on the HDG-machine[11] and developed *evaluation transformers* as a way to specify *evaluation degree* of expressions [137, 46].

Further notable work revolved around Multilisp [107], a parallel Lisp implementation, and Mul-T RTS for parallel Scheme [140], which was among the first to describe the concept of futures and lazy task creation [183], which allows tasks to be uninlined if parallel execution is desired. Distributed Filaments investigated exploiting fine-grained fork/join parallelism on multiprocessors, using stateless threads [95].

Examples of languages with implicit parallelism include Id [10] and pH, introducing the use of I-Structures [187] and M-Structures [20], shared data structures for synchronisation that are either empty or full, and can be used to implement futures. Performance competitive to Fortran was demonstrated by SISAL [82, 51] using the Livermore Loops benchmark, an applicative array- and stream-oriented language for HPC and numerical applications.

Additionally, the implicitly data-parallel NESL [36] introduced the flattening transformation to improve performance. The Manticore [86] implementation of Concurrent ML provides support for implicit data-parallelism similar to NESL and explicit message passing, a non-deterministic choice operator, futures, and explicit synchonisation.

Single-Assignment C (SAC) [213, 214] is a first-order array programming language with implicit parallelism, focused on high productivity through high level of abstraction, and high performance through exploitation of parallelism over first-class n-dimensional arrays with potential for delayed type specialisation. This makes the

[11] i.e. the Highly-Distributed G-machine

language well-suited for applications in signal processing and numerical computing, as well as for execution on GPUs and FPGAs. The performance achieved by the optimising compiler was shown in many cases to be similar to that of hand-optimised code.

Haskell proved an exceptionally fertile base language for many different parallel EDSLs [166]. Among those, most notable include Par Monad [168] and Evaluation Strategies discussed in Chapter 3. Par Monad is implemented as a library and uses Haskells concurreny support to provide deterministic parallelism using `IVars`, which are inspired by I-structures. Parallelism is explicitly expressed using the `fork` function that creates a new lightweight thread.

Eden [164] extends Haskell and its RTS with explicit process creation primitives and channels that are used to communicate data fully evaluated by parent process as arguments to processes, whilst processes produce the outputs eagerly. Instantiated processes are executed in parallel using a shared-nothing message passing model. Communication and synchronisation are handled implicitly by the RTS. HOFs are used to define skeletons on top of the basic language primitives, which encapsulate common patterns of parallelism [29, 28].

A recent paper compares parallelism features of Haskell, F#, and Scala [232] using the N-Body application, providing an overview of a high-level approach to parallelism in popular functional languages. Parallel and distributed Haskells are compared by Trinder et al. [239]. Implictily parallel PMLS (Parallel ML with Skeletons) is compared to GpH and Eden by Loidl et al. [161].

Instead of implementing a whole new language and the associated toolchain, another approach is to use existing language features to implement *libraries* that support parallelism. It is possible to exploit parallelism at process level which requires adding the ability to coordinated execution across OS processes. One example is PThreads [186], which is a library conforming to the POSIX standard that implements threads and offers synchronisation primitives that can be used to implement higher-level abstractions. Low-level libraries such as PVM [98] and MPI [105] (e.g. MPICH and OpenMPI) provide the means to explicitly arrange distributed com-

putations by using the provided communication abstractions, i.e. send/receive and collective operations.

Higher-level libraries provide abstractions such as skeletons, which can be implemented using higher-order functions. These encapsulate patterns of parallelism and coordination and can hide most complexity from the library users, who are only required to plug in a computation, since parallelism is transparently managed by the skeleton implementation [103].

A more refined sub-division of applications may be possible based on the used parallelism patterns, which in higher-order languages coincide with the notion of skeletons. The `map` function, which applies a function to each element of a collection simultaneously, can be parallelised and is an example of data parallelism, and thus can be considered a pattern. In a skeleton-based language parallel map can be implemented using a task farm approach with either static or dynamic scheduling, where a master process dispatches work to the workers.

The `fold` function, which reduces a data structure using a binary operator, can be parallelised and implemented efficiently using a tree structure. Note that this pattern is very general [129]: for instance it can be used to implement `map`.

Another pattern is the *pipeline*, where multiple stages are arranged and data is streamed through the pipeline. Semantically, it is similar to function composition and is parallelised for stage to work on different data at the same time.

Other patterns include `zip` that takes two lists of same length and returns a list of tuples, `zipWith` which acts as `zip` but takes a binary function and applies it to tuple elements to return a list, `scan`, task farm, parallel workpool, stencil, and rolling buffer, among others.

Further notable patterns include divide-and-conquer, where a problem is recursively sub-divided into smaller problems until those can be solved in parallel and the combined to form the final result, and branch-and-bound [6] which implements a search where paths can be abandoned as soon as it becomes clear the the best solution can not be found using that branch.

We review Glasgow Parallel Haskell (GpH) and a HOF-based approach to par-

allelism in GpH called Evaluation Strategies, as well as the GUM RTS in the next Chapter. More details about different programming models can be found in a recent technical report [24].

2.2.7 Applications

A recent publication reviewed the success stories related to and the increasing popularity of functional languages, concluding that it "had a wide impact on the society as a new generation of programming" [121].

Apart from wide uses in research and in education, the article quotes many applications across different domains: symbolic computation and compilers as historical core domains, WhatsApp using Erlang to power its messaging servers, and Facebook using Haskell to improve news feed performance and to filter out spam. Other companies including LinkedIn and Skyscanner use Scala in the backend, whilst Google's highly successful MapReduce was inspired by the HOFs `map` and `reduce`. Moreover, functional EDSLs have been successfully used in circuit design [33].

As verification is facilitated through equational reasoning, applications that require high confidence in, or even proof of, the correctness of the software, such as in the financial and defense sectors, functional programming is being employed on a large scale. Notable companies include Galois, Jane Steet Capital and QuviQ.

2.3 Implementing Parallel Functional Languages

This subsection reviews the history and state-of-the-art in implementation of parallel functional languages and includes distributed-memory as well as shared-memory implementations. The focus is on approaches to evaluation, covering eager, lenient, and lazy evaluation, as well as on abstract machines and distributed graph reduction.

2.3.1 Approaches to Evaluation

To implement strict or non-strict semantics that were introduced in Subsection 2.2.5, an evaluation model needs to be chosen that defines how expressions are evaluated,

for instance eager evaluation for the former or lazy evaluation for the latter.

To evaluate an expression such as `f` e_1 ... e_n, *eager evaluation*, also termed *call-by-value*, will first evaluate all the arguments to the function `f` before proceeding to evaluate the function body.

By contrast, *lazy* evaluation using normal order reduction will first begin to evaluate a function body by passing the arguments unevaluated and deferring their evaluation until the demand is expressed. Lazy evaluation, also referred to as *call-by-need*, will thus never perform unnecessary work. However, it is at odds with the idea of exploiting parallelism as it aims to delay evaluation of arguments as long as possible, although it is imaginable that values of multiple expressions are demanded simultaneously. Thus strictness analysis can be used to determine evaluating which arguments should not be delayed [64].

A middle ground is sought by *lenient evaluation* that seeks to retain expressiveness of non-strict semantics, whilst exploiting as much parallelism as if using eager evaluation, by evaluating both function body and arguments in parallel, but only as far as data dependencies allow [233]. Moreover, lenient evaluation appears a good match for dataflow languages with massive fine-grained parallelism [10] and for speculation, where some parts of the computation are attempted optimistically and are cancelled, or the results are discarded once it becomes clear that they are not required.

Although in the lazy setting the implementation is complicated through the need for strictness analysis and maintenance of thunks [37] that represent delayed evaluation, normal order reduction is guaranteed to produce the result if such exists and laziness avoids potential work duplication through sharing, whilst improving expressiveness and modularity [244, 126] by facilitating separation of data from control. As such it appears a good fit for pipeline- and stream-oriented parallelism, where data are gradually processed, avoiding the full instantiation of the whole stream. In particular, a program may be split into generators of potentially infinitely many candidate results and a selector that chooses the suitable one.

2.3.2 Abstract Machines

An *abstract machine* defines how programs written in or compiled to this machine's instruction set are evaluated step-by-step, usually with the help of a stack, a store, and registers. Omitting many details of real hardware, abstract machines are suitable as an intermediate target for interpretation or compilation. We focus on abstract machines for functional languages, whilst more information can be found in an annotated bibliography by Diehl et al. [75].

The first abstract machine for evaluating λ-calculus expressions was Landin's SECD machine [145] (short for stack, environment, control, and dump), also used as a target for ISWIM. One of its key novelties was the use of *closures* in the heap to represent functions, but it also left many operational aspects unspecified. SECD was extended by Cardelli [52] and adapted for non-strict languages by Burge [45] and Henderson [115]. SECD-M extends the original SECD machine with concurrency and non-determinism [1]. More details can be found in Chapter 10 of the textbook by Field and Harrison [84].

In his seminal paper Turner describes compilation of SASL to a fixed set of combinators, which correspond to graph rewriting rules [253, 243], combining ideas from combinatory logic with λ-calculus. The the SK-machine provides instructions the S and K combinators, with derived instructions equivalent to a number of the key additional combinators, added for efficiency. Johnsson and Hughes showed the benefits of using more coarse grained combinators derived through λ-lifting [134] and as supercombinators [127, 128], respectively.

Further developments and optimisations led to compiled graph reduction and the G-machine [15], originally used for evalution of Lazy ML programs. Subsequent extensions include the Spineless G-machine [47] that limits updates to shared expressions significantly reducing the amount of heap and stack accesses. The Spineless Tagless G-machine [195] (STG) uses a uniform representation of objects on the heap as closures with a code pointer in the first field used as a direct jump avoiding the need to allocate and examine an extra tag field. It uses a small functional language as its intermediate language. It is the basis of the current GHC Haskell imple-

mentation [201, 169, 170]. Tags were later reintroduced for efficiency on modern architectures, where the tagless scheme negatively impacts branch prediction [171].

An alternative approach is taken by the Categorical Abstract Machine [69] that uses instructions modelled on the Cartesian closed category and combinatory logic. This is used in the implementation of Caml, and its more efficient ZINC [152] and Caml Light [153] variants, forming the basis for OCaml [154].

The related Krivine machine [141] is a three-instruction abstract machine for call-by-name evaluation (i.e. without sharing) of the λ-calculus, using the argument stack also as a return stack (i.e. as continuation). It can be extended to implement call-by-need evaluation [218]. Similarly, the Three Instruction Machine (TIM) has three instructions supporting lazy supercombinator reduction [81, 7] and is known for pioneering the uniform representation design inspiring tagless implementations. An issue for lazy parallel implementation is that the stack frame includes arguments that may not be needed exacerbating the overhead of graph shipping and requiring more remote pointers in a distributed setup [111].

2.3.3 Parallel and Distributed Graph Reduction

Large scale parallel machines often span clusters of nodes and require distributed execution to exploit all the available PEs to improve scalability and performance. This introduces many challenges with respect to implementation of the coordination aspects mentioned above, which we discuss in the next subsection. In this subsection, we focus on distributed graph reduction and abstract machines that were created with distribution in mind. A more detailed abstract machine comparison can be found in Hammond and Michaelson [111].

Lower-level abstractions such as Partitioned Global Address Space [254] require the programmer to control the mapping of data to PEs explicitly, which reduces productivity, but may help boost performance by improving data locality. Similarly, the Actor model [119, 2], as used in Erlang and Scala, and coordination languages, e.g. Linda [99] and Caliban [136], build on top of a communication layer (message passing), offering foundation for higher-level abstractions.

Several parallel reduction machines have been proposed. In most cases, a sequential machine is used on each node or PE and is extended with a coordination protocol and potentially a shared store abstraction, which presents additional challenge of efficient and scalable implementation.

The v,G-machine [14] follows a packet-based approach. It uses tags, where each packet represents a closure and also has a local stack. It uses sequential compilation technology and a harness that manages communication and starting up the sequential G-machines on each PE, whilst sharing the graph and the task pool.

The Four-Stroke Reduction Engine [65] is a supercombinator reduction machine that interprets the program graph and was the first abstract machine to be implemented on the GRIP architecture. It is notable for introducing many concepts and mechanisms that influenced further development of parallel graph reduction, for instance the "evaluate-and-die" evaluation model (even though it was not so termed at the time) with implicit communication and encoding of the dump in the heap rather than using an explicit stack, among others [111].

The Highly-Distributed G-machine (HDG) [137] is based on the Spineless G-machine that uses a stackless design and was developed for a network of Transputers utilising the evaluation transformers reduction model. This demonstrated speedups on several small benchmark applications, offering evidence for the feasibility of executing implicitly parallel functional programs on distributed architectures despite the significant communication costs.

The STG-machine is designed to be suitable for parallel evaluation [198] and is, with many optimisations, used as the basis for the current shared-memory GHC-SMP runtime system [169], as well as for the distributed GUM RTS, discussed in more detail in Chapter 3. A variant of the STG machine is also used in the DREAM RTS [41] for the Eden language as it shares a large part of the RTS with GUM and extends it with implict channels and support for zero-copy communication on shared-memory machines. Recently, GHC's SMP RTS and GUM have been combined to yield a multi-level RTS, which allows balancing between the distributed and the shared heap [3].

2.4 Adaptive Control of Parallelism

In this subsection we review common approaches to parallelism control with focus on adaptivity as found in several original mechanisms and recent extensions. We discuss key RTS mechanisms [27, 30], starting with load distribution strategies, moving on to scheduling, memory management including virtual shared memory, followed by communication, and granularity control. As most coordination-related decisions are NP-hard, as for instance the locality-maximising placement that can be mapped to an instance of the Bin Packing problem [60], most of the mechanisms discussed are heuristics-based and lead to sub-optimal solutions in most cases, which is additionally exacerbated by probabilistic operational behaviour of the baseline work stealing algorithm[12].

2.4.1 Load Distribution

Load distribution is one of the key areas of parallelism management as it involves the trade-off between spreading the work across the PEs to balance the computational load to increase utilisation, whilst grouping related data and computations to ensure locality and avoid communication overhead. There are two main classes of mechanisms: work stealing and work pushing.

Load distribution may be static or dynamic. We focus on dynamic load distribution, as static distribution is too inflexible to be considered adaptive. However, in some cases using static load distribution is justified if all knowledge for reaching the optimal distribution is available at compilation time. In lazy distributed graph reduction machines, load distribution is dynamic, because parallelism is created as evaluation unfolds.

Work stealing is a demand-driven load distribution mechanism [260] popularised by the work in the context of Cilk [38], but was commonly used in prior implementations of parallel functional languages mentioned above [49, 140, 194]. Here idle PEs attempt to steal work from busy PEs, thus minimising the amount of requests when load in the system is high. In most implementations the victims of steal attempts

[12] whilst value-determinism of the result can still be maintained

are chosen at random, which does not require global knowledge and therefore constitutes a scalable design. However, many stealing requests are created when the system is lightly loaded, as idle PEs will attempt to initiate work transfer. One way to prefetch work is to use watermarks – if a PE is about to run out of work it may start searching for work before this actually happens in the hope that new work will have arrived by the time it actually runs out of work.

By contrast, *work pushing* is a sender-initiated way to distribute load, where the busy PEs attemt to offload work to less busy PEs [79]. This potentially generates less messages, but tends to create many messages when the system load is already high, which may overwhelm the system.

An interesting result for the job-shop formulation of load balancing problem, based on choosing two workers, interrogating them about their load and then passing the load to the less loaded one, results in a close-to-optimal decision [182].

Other scalable algorithms include *gossip* (or epidemic broadcast) [73], where the information or load is shared to the immediate neighbours. This is similar to a diffusion model of communication as used in a distributed implementation of graph reduction for a hypercube architecture [101, 100], although such protocol may be slow to react to frequent load changes.

Both work pushing and work stealing can be considered adaptive as they respond to changing load levels and parallelism degree, although some implementations exhibit additional adaptivity by taking into account additional system parameters such as task granularity if available [131]. For instance, the BUSD mechanism [173] uses breadth-first (FIFO) distribution until saturation and then switches to depth-first (LIFO) on a Transputer-based implementation of the ZAPP architecture [173]. Whilst using architectural parameters may be beneficial we consider them static. In any case, some limiting back-off mechanism and restriction on the number of messages per PE is commonly used to avoid flooding the system with messages.

2.4.2 Scheduling

We refer to scheduling in a narrow sense as dispatching light-weight threads on a single node under RTS control, which is similar to OS-level scheduling of processes and heavy-weight threads [221]. We focus the discussion on dynamic rather than static scheduling [217, 35].

Scheduling includes thread management, such as creation of threads, layout of thread descriptors, as well deciding which of the runnable threads to execute and how to implement a thread pool. In particular, multi-threading is needed for asynchronous execution that helps to hide communication and I/O latencies by executing a runnable thread, whilst other threads may be blocked waiting for messages.

Scheduling can be classified as *fair*, when each thread is guaranteed to run for some time and will not be starved, or *unfair*, where theoretically a thread can't be prevented from running indefinitely. Although desirable, fairness may incur additional book-keeping and context-switching overheads.

Scheduling may be *pre-emptive*, where the scheduler may interrupt threads and re-schedule. Or scheduling may be *cooperative*, where the running thread is required to yield once it has finished execution.

For instance, a pre-emptive round-robin mechanism can be considered fair as each thread can work in turn for the same specified time slice using the same resources. However, a priority-based scheme can be unfair if low-priority tasks end up being postponed indefinitely, if new high-priority threads are continuously spawned [221].

Thread management is more complicated if parallelism in the language is advisory, due to the need to efficiently manage potential threads that repserent parallelism, but have not yet been turned into full-blown threads. In particular, if parallelism is very fine-grained, as in the functional setting, management of potential parallelism has to be very light-weight. We will discuss creation of potential parallelism, also termed *sparking*, in more detail in Chapter 3. By contrast, mandatory thread creation leads to immediate allocation of the thread descriptor, but gives up the flexibility to ignore the parallelism even if it is beneficial to do so, for example to dynamically increase thread granularity [183].

The notification model can be either *synchronous*, when either the child thread notifies the parent once the computation is finished, or a barrier synchronisation is periodically used as with fork-and-join parallelism. Or a notification model may be *asynchronous* where synchronisation is implicit via the graph, where node of the graph representing the value to be computing is replaced by the value once available and only those threads waiting on that value are notified. The asynchronous approch is potentially adaptive to load as the first thread to access a graph node, potentially its parent, will claim and evaluate it.

2.4.3 Memory Management

Managing memory allocation, and heap and stack layout, can be challengening especially if performed manually by the programmer. Automated garbage collection (GC), which releases the previously allocated memory once it is no longer in use, shifts the responsibility to the implementation [176], and is common with most high-level and, in particular, declarative languages. There are several main GC mechanisms: reference counting, mark-and-sweep, copying (two-space), and generational GC [135].

As the name suggests, reference counting keeps track of the number of references to each object in the system and safely releases an object once the count has gone down to zero. Despite the conceptual elegance, the overhead of incrementing and decrementing the counts is significant and the scheme has difficulties with detecting and collecting cycles.

Instead, a mark-and-sweep scheme traces whether the objects can be referenced in the heap from the root set and collects the unreachable ones. This scheme is capable of collecting cycles, but often requires the execution (mutation) to be halted entirely, while GC is being performed (in the so-called "stop-the-world" variants). Furthermore, the GC time tends to depend on the heap size.

Another scheme is a copying (two-space) collector that copies reachable object to the new space and thus compacts the memory and frees up space in the old-space at the cost of copying. After the operation is complete, the new space becomes old

and the old is now the new where allocation happens.

Based on the insight that few old objects that have already survived for a long time will remain alive and most objects expire early, a generational collector has a nursery space, where new objects are placed, which are then promoted to older generations that are collected less often, if they have survived until the collection phase.

Distributed GC is needed if parts of the graphs are shared across remote PEs and adds to the challenge through the need to track inter-PE references [203], effectively creating a (partial) virtual shared memory view of memory, for instance using a distributed reference counting scheme [31], even though cycles can't be collected in this scheme. GC can be considered adaptive with respect to object use patterns, and in specific implementations can change the heap size at run time [9].

2.4.4 Communication

In general, communication including serialisation and message packing accounts for a substantial portion of the overhead in the distributed setting. One of the key decisions for a graph reducer, which handles communication and synchronisation implicitly inside the RTS, is much sub-graph to pack into a packet [160]. The choice is between packing only indirections that may prompt further communication later on, or some amount of graph. It may be beneficial to pack some neighbouring graph as well to implement a version of prefetching as it is likely to be required too. On the other hand, a decision could be made to only send normal form data, as done in Eden, which avoids the need to maintain the virtual shared memory abstraction.

Latency hiding can be implemented using multi-threading, where packing and sending of the message buffer is performed by a separate thread. The trade-off is between local evaluation where only an indirection is sent off, and remote evaluation where the expression and the needed data are sent off [157].

Another decision is whether to treat communication as a priority task, before computation is performed. Moreover, if thread migration is to be supported, a more complicated packing scheme needs to be implemented [249]. In the context of

a non-strict distributed functional RTS (extended version of GUM), it was shown that limiting the export based on the network hierarchy level can be beneficial for performance in hierarchical clusters [13].

2.4.5 Granularity Control

Granularity of a thread informally refers to the "size" of the associated computation, which is often abstractly measured in clock cycles, or more concretely in mutation time, representing the amount of work associated with that thread. Related to partitioning, granularity control is one of the key decision areas for the RTS [157], and for the programmer if annotations or explicit thresholds are used [125, 188, 159].

Granularity control aims to resolve a trade-off between increased overhead of parallelism managment, if granularity is too fine, and limited parallelism, if granularity is too coarse. As predicting granularity is challenging in general, especially for recursive and higher-order functions [100], due to its dependence on architecture as well as on the application characteristics, some flexibility is desirable to allow the RTS to adapt granularity at run time.

A common explicit way to specify granularity is through programmer annotations or specified thresholds, such as a limit on depth in divide-and-conquer computations which determines whether a sub-problem should be sub-divided and solved in parallel or solved sequentially. Such a crude approach has been shown to perform well for balanced D&C applications, whereas it exhibits limited flexibility when the compute tree is unbalanced. More sophisticated algorithms use a notion of fuel that is dynamically distributed as computation unfolds, and of give-back, where unused parallelism instead of being discarded may be used later on, exploiting circularity in data structures, which requires laziness in the language to express these schemes at library level [231].

Alternatively, granularity control can be built into the RTS implementing lower-level parallelism control. Using mandatory parallelism would result in *eager thread creation*, whilst advisory parallelism in using more flexible, but more costly *evaluate-and-die* model of computation, which is similar to lazy task creation [183]. Potential

parallelism is handled, either by creating parallelism, but either leaving the option to inline it back into the parent thread if necessary [183], or by first seeding parallelism, and later un-inlining it, if additional resources have become available [211, 102]. More coarse-grain granularity tends to reduce the number of threads and thus the parallelism overhead, but care must be taken to prevent starvation and load imbalance by avoiding too coarse settings. Using granularity information in GpH was shown to improve work stealing performance on hierarchical clusters [131]. More information, in particular on compile-time granularity analyses, can be found in Loidl's thesis [157].

2.4.6 Run-Time System Comparison

Table 2.1 provides an overview of GUM compared to the most recent related systems, which together span a wide spectrum of parallel language run-time systems. For more detailed and broader comparisons refer to further literature [24, 23].

With respect to parallelism identification GUM and SMP occupy a unique place in the design space as the annotations provide hints that are advisory rather than mandatory, as is e.g. process instantiation performend in an Eden program, which will lead to a creation of a remote process. Eden and GUM are similar in the architectural respect that unlike other systems they enable distributed execution. On the other hand they differ in the implementation as GUM provides a Global Indirection Table for inter-PE pointers implementing the virtual shared memory abstraction, whilst DREAM uses shared-nothing design and sends data once it is in normal form. Manticore and X10 are somewhat similar in chosing to incorporate both implicit data parallelism and explicit task parallelism, whilst GUM makes no special arrangements for data parallelism and treats expressions requiring data as tasks.

There is no agreement on the scheduling style among the systems, Manticore allowing nested schedulers and X10 following PGAS distribution style. GUM and SMP follow the evaluate-and-die model that leads to an unfair design, but helps improve performance.

Table 2.1: Overview of GUM and Related Systems

RTS (Language)	parallelism identification	scheduling	architecture	synchronisation	load balancing
Cilk [38] (C ext.)	explicit (`cilk_spawn`)	LIFO	shared	explicit	work stealing
GHC-SMP [169] (GpH)	annotations (advisory)	FIFO unfair	shared	implicit	work stealing
Manticore [86] (NESL/CML-alike)	impl. data par. expl. task par.	FIFO nestable	shared	implicit	work pushing
X10 [54] (X10)	impl. data par. expl. task par.	PGAS	shared	implicit	work stealing
GUM [237] (GpH)	annotations (advisory)	FIFO unfair	virtual shared	implicit	work stealing
DREAM [41] (Eden)	explicit process instantiation	round robin fair	shared-nothing	implicit	work pushing

In all these systems, thread and memory management are implicit as well as synchronisation, with an exception of Cilk. This allows for a high level of expression, compared to explicit synchronisation and parallelism management. Despite the popularity of work stealing, some systems have chosen to use work pushing to reduce the amount of communication. This diversity exacerbates the difficulty of directly comparing these systems and languages. Thus, we will focus on GUM and SMP, and to a lesser extent DREAM as they are most closely related.

2.5 Summary

This chapter presented an overview of the literature and of the state-of-the-art in areas related to parallel functional programming in general, and adaptive RTS-level parallelism control for distributed execution of non-strict parallel purely functional languages in particular. In particular, this Chapter provides context for the detailed discussion of the GUM RTS for GpH in the next Chapter, as well as points to related work.

First, we have seen how architectural trends, driven by fundamental physical limitations, have made a turn towards increasingly more hierarchical, heterogeneous and massively parallel architectures. The main distinction is in the memory arrangement: whether the memory is shared across all PEs, or each PE owns a private mem-

ory and requires communication to exchange data between PEs. There is evidence that the shared-nothing design is more scalable, but higher performance can potentially be reached if expensive communication over a slow network can be avoided. At rack scale, which constitutes an intermediate point in the design space between multi-cores and globally distributed multiprocessors, substantially faster communication networks can be employed that allow scalability beyond a single node, but keep communication costs lower and provide higher throughput (e.g. Infiniband).

Next, we followed the development of parallel programming models and traced the evolution of functional languages to provide means to exploit parallelism offered by the underlying hardware. Automatic parallelism has turned out to be an elusive goal so far, but we have demonstrated the benefits and trade-offs associated both with low- and high-level models. We have argued that manual tuning often exacerbates portability and maintenance issues and therefore the advantages of the high-level approach gain in importance.

Subsequently, we have seen that substantial challenges are associated with the implementation of efficient parallel run-time systems which can support parallel execution in ways in which libraries might not and are usually based on an abstract machine as an intermediate target. In particular, to make use of additional flexibility requires careful parameterisation and heuristics choice that reflect the architecture-system-input combination well enough. This is exacerbated in case where languages use non-strict semantics and are implemented with distributed graph reduction.

Finally, we reviewed the RTS-level parallelism control mechanisms for load distribution, scheduling, memory management, communication and granularity with focus on adaptive variants of the mechanisms as they exhibit added flexibility for distributed execution of semi-explicit parallel functional languages, comparing GUM to related run-time systems.

Chapter 3

Graph Reduction on a Unified Machine Model

The GUM (Graph Reduction on a Unified Machine Model) [237] run-time system (RTS) for Glasgow parallel Haskell (GpH) [235, 238] implements distributed graph reduction supporting execution on shared- and distributed-memory architectures. This chapter describes GUM's design and the driving forces behind it, its abstract model, and the implementation of the key components, policies and mechanisms, such as load distribution.

The focus of this thesis is on extending GUM's adaptive policy control mechanisms to increase performance and scalability on distributed-memory platforms by improving load balancing and data locality. Finding optimal policies and policy parameters to achieve high performance is challenging, since parallelism is automatically controlled by the RTS. Hence the discussion is focused on the extended system components that implement advisory parallelism, profiling, scheduling and load balancing using work stealing.

3.1 Language Overview

This section provides an overview of the semi-explicit programming model and syntax of GpH for identification of parallelism. We also briefly introduce Evaluation Strategies [236, 167] used to implement abstractions that separate computation and

coordination concerns on top of the basic parallelism primitives provided by GpH.

Then graph reduction and the Unified Machine Model are discussed, which provide substantial flexibility for adaptive architecture-transparent management of advisory parallelism.

3.1.1 Haskell Extension for Semi-Explicit Parallelism

Glasgow Parallel Haskell (GpH) [235, 238] extends Haskell [172, 123], a de-facto standard non-strict purely functional language, by adding both sequential and parallel combinators, `par` and `pseq` of type `a → b → b`. These combinators allow the identification of potential parallelism as well as evaluation order and evaluation degree. Both combinators are projections onto their second argument, i.e. they return the result of evaluating the second argument. This way GpH remains a *conservative* extension of sequential Haskell retaining referential transparency [235]. The programming model is high-level and *semi-explicit* because, apart from identification of potential parallelism and evaluation degree and order, other coordination aspects, such as communication and synchronisation, are implicitly controlled by the RTS.

The `pseq` and `par` combinators are implemented as *built-in functions*[1]. These are translated by the compiler to a call within the RTS. The multi-threaded GHC-SMP RTS [169] supports execution on shared-memory multi-cores, whilst GUM (or GHC-GUM) supports execution on distributed-memory clusters of multi-cores.

Using `par`, the programmer provides a hint to the RTS that the first expression can be beneficially evaluated in parallel by creating a *spark*. A spark is a pointer to a sub-graph, represented by `par`'s first argument, which can be reduced in parallel. Compared to thread creation, sparking is cheap as it amounts to adding a pointer to the *spark pool* data structure. Ultimately, the RTS decides whether the spark will be 1) turned into a light-weight thread if no runnable threads are available, 2) inlined into its parent if the spark was not picked up before parent subsumes it, or 3) discarded if the spark pool is full or spark refers to an already-evaluated expression. We discuss management of threads and sparks in more detail below.

[1] the functions can be bound to variables and passed to functions, which is exploited in the implementation of the Evaluation Strategies

Note that to create *useful* parallelism the first expression supplied to `par` has to satisfy the following conditions [166]:

- it is unevaluated and not already under evaluation,
- it represents a large-enough computation relative to the associated overhead,
- it is not immediately required by the parent thread (if it would be immediately required then it could be inlined into the parent, i.e. executed by the thread that created the spark, and thus not introduce any additional parallelism),
- it is shared with the rest of the program to avoid being reclaimed by the GC.

This mechanism can be viewed as implementing *lazy futures* [107, 140] as the computation is lazily started when the result is demanded instead of being eagerly evaluated at the spark creation time, where a future (sometimes also called a *promise*) refers to an eventually available value (a proxy for a result) [97, 17, 156]. The implementation may support implicit synchronisation.

The parallelism is *advisory*, because the RTS is free to ignore the hint and evaluate the expression sequentially. This better supports adaptive policy control by offering more flexibility than the common approach where parallelism is *mandatory*, as threads are created explicitly at every fork/spawn site. Additionally, it is cheaper to exchange sparks than threads between PEs.

```
1  fib :: Integer -> Integer
2  fib 0 = 0                              -- sequential version
3  fib 1 = 1
4  fib n = fib (n-1) + fib (n-2)
5
6  pfib :: Integer -> Integer -> Integer
7  pfib 0 _ = 0                           -- parallel version
8  pfib 1 _ = 1
9  pfib n t | n <= t    = fib n           -- t for granularity tuning
10          | otherwise =  x `par` y `pseq` x + y
11            where x = pfib (n-1) t
12                  y = pfib (n-2) t
```

Listing 3.1: Sequential and Parallel Fibonacci Functions in GpH

Listing 3.1 provides an example of how a sequential function `fib` is modified to enable parallel execution. The changes are fairly small and require introduction of `par` to identify parallelism and of `pseq` to specify evaluation order. Thus in line 10, first `x `par` y` is evaluated creating a spark for `x` and returning `y` and then the sum `x + y` is evaluated thus demanding the results of both sub-expressions. Additionally, a threshold `t` is used for application-level granularity control by restricting sparking to the specified depth in the compute tree. The inherent parallelism is too fine-grained to merit full exploitation due to the relatively high per-thread overhead. Granularity control is discussed in more detail in Section 3.3.2.

Although the changes to the code in Listing 3.1 are minor, this is a somewhat unstructured approach as *coordination* and *computation* aspects are intermixed. In larger programs, this is unsatisfactory as the algorithm is obscured by coordination aspects which are spread across multiple locations in the code, making the code more difficult to read and maintain.

3.1.2 Evaluation Strategies

To cleanly separate the computation and coordination concerns, *Evaluation Strategies* [236, 167] were introduced on top of the basic primitives to facilitate understanding of the algorithm without considering the coordination aspects as well as allowing changes to coordination without the need to change the computation.

Listing 3.2 below shows the definition of the `Eval` monad, which allows to separate operational aspects of coordination from pure computations, as well as some basic Strategies. For instance, `parList` applies a strategy to each element of a list in parallel using the sequential `evalList`. In turn, `parList` can be used to implement the `parMap` skeleton, which applies a function to each element of the list in parallel.

One challenge is choosing optimal granularity by tuning the degree of evaluation. This can be achieved by using an appropriate Evaluation Strategy – an expression of monadic type `Eval` that takes another strategy that determines evaluation degree as an argument. Listing 3.2 illustrates the core data types and basic Evaluation Strategies that can be composed to form more complex parameretised ones.

```
data Eval a = Done a
type Strategy a = a -> Eval a
runEval :: Eval a -> a
runEval (Done x) = x

instance Monad Eval where
  return  = Done
  m >>= k = case m of Done x -> k x

r0, rseq, rpar :: Strategy a
r0   x = return x             -- no evaluation
rseq x = x `pseq` return x    -- evaluate to whnf
rpar x = x `par`  return x    -- create a spark

rdeepseq :: NFData a => Strategy a  -- evaluate to normal form
rdeepseq x = x `deepseq` ()         -- relies on Control.DeepSeq

using :: a -> Strategy a -> a       -- strategy application
e `using` strat = runEval (strat e)

dot :: Strategy a -> Strategy a -> Strategy a    -- composition
strat2 `dot` strat1 = strat2 . runEval . strat1

-- applies its strategy argument to all elements of a list
evalList, parList :: Strategy a -> Strategy [a]
evalList strat []     = return []
evalList strat (x:xs) = do x' <- strat x
                           xs' <- evalList strat xs
                           return (x':xs')

-- applies a strategy to all elements of a list in parallel
parList strat = evalList (rpar `dot` strat)
-- a parallel map skeleton
parMap :: Strategy b -> (a -> b) -> [a] -> [b]
parMap strat f xs = map f xs `using` parList strat
```

Listing 3.2: Basic `Eval` Strategies (from [167])

For instance `rdeepseq` can be passed in to demand full evaluation to NF by recursively applying the strategy, thus forcing evaluation. In particular, note how `parMap` skeleton is implemented using `parList`, which applies its strategy argument to all elements of a list in parallel by sparking each with `rpar`, which is in turn passed as argument to `evalList`. The `strat` argument determines the evaluation degree for each list element. Also note the separation of algorithm and strategy with `using` in line 35.

Design decisions and implementation details of this new version of the Evaluation Strategies can be found in the literature [167, 166].

3.1.3 Graph Reduction

Graph Reduction is an implementation technique [250] and evaluation model for functional languages based on the λ-calculus [62, 19, 18]. As shown in Figure 3.1, the computation is logically represented through a graph of operations (functions/combinators) and values[2]. The reduction process describes the application of functions to their arguments and overwriting the application node with its result, to eventually obtain the final value as a result. The example from [157] shows how the result value of `49` is computed from the expression `square (1+2*3) where square x = x*x` by repeated function application. Note that the value 7 bound to `x` is computed only once and then shared.

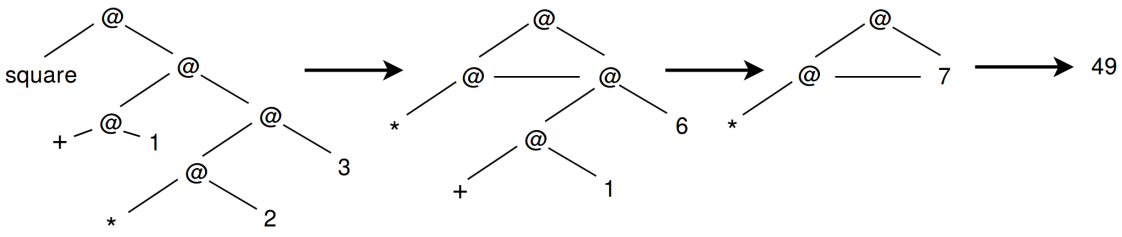


Figure 3.1: Graph Reduction Example

The corresponding computational structure resembles a dynamically expanding and contracting general graph as opposed to a directed acyclic graph (DAG), due to recursion and because sub-expressions may be shared to avoid work duplication.

[2]the actual implementation may use a different data structure for efficiency

Choosing appropriate evaluation degree affects the potential parallelism.

In a lazy language, the default evaluation degree is to weak-head normal form (WHNF). This means that only the outermost constructor or function is applied, potentially resulting in too fine granularity (cf Section 3.3.2). By contrast, full evaluation to normal form (NF) as used by strict languages leads to reduction until no reduceable expressions (redexes) remain.

However, the choice of evaluation degree is not binary – it may be beneficial or feasible to evaluate only a part of a large and complex data structure, for example when the full data structure is too large to fit into the local memory available. Moreover, tuning the actual degree of parallelism as a fraction of potential parallelism (e.g. through thread subsumption or application-level explicit thresholding, chunking or clustering [162]) can help increase granularity and reduce overheads.

Additionally, the evaluation model may be chosen from the following and influence the implementation of RTS policy control. Common evaluation models are *strict*, *lazy* or *lenient*, as discussed in Section 2.3.1. Under strict evaluation, the function is applied to its arguments once the arguments have been evaluated to NF, whilst under lazy evaluation, a thunk is created to represent yet unevaluated expression which is by default evaluated to WHNF and only fully evaluates its arguments when they are demanded. Whilst strict evaluation simplifies speculative and parallel execution and avoids the need to use thunks, in some cases it may result in unnecessary work duplication that is avoided in the lazy (demand-driven) setting, and a strict program can crash or fail to terminate[3], where a lazy program would terminate delivering the final result. It is possible for a language to support both strict and lazy evaluation and annotations as well as strictness analysis can be used to help predict program's needs. Furthermore, lenient evaluation can provide a flexible middle ground, as discussed in Section 2.3.1, where a function body can be evaluated in parallel with its arguments. This can potentially result in an evaluation degree somewhere inbetween WHNF and NF.

Lazy evaluation is one of the main driving forces behind the design of GUM since, due to lazy evaluation, thunks need to be maintained for potentially shared

[3]e.g. in `take 5 [1..]` or `fst (1, error "boom!")`

computations that have not yet been evaluated. Thus we need to ensure that in our program the results of a sparked computation are demanded. Strictness annotations can be used to avoid deferred evaluation of values that are known to be required.

3.1.4 Unified Machine Model

A key feature of GUM is its machine-independent execution model that uses a network of abstract processing elements (PEs) with private heaps that interact via logically sending and receiving messages, which allows for execution on both shared-memory and distributed-memory architectures. Whilst physical memory is not shared in this model, there exist inter-heap pointers maintained using a *virtual shared memory* implementation to allow sharing of sub-graphs across PEs to avoid work duplication.

Notably, GUM's virtual shared memory implementation hides explicit communication from the programmer, whilst enabling execution on distributed-memory architectures. It allows thunks and their values to be shared across multiple PEs. This enables GUM's deterministic unified programming model for both shared-memory and distributed-memory architectures, whilst avoiding race conditions and deadlocks. Note that the determinism refers to the final result, rather than operational behaviour, which may differ from run to run, as some RTS decisions are randomised.

Moreover, the key adaptive mechanisms of thread subsumption, as discussed in Section 3.3.1, and work stealing, as discussed in Section 3.3.4, are very generic and align well with the architecture-independent approach. Thread subsumption in conjunction with advisory parallelism removes the need to specify a particular number of threads in relation to the number of PEs. In turn, work stealing only relies on local knowledge and operates in a decentralised fashion avoiding static communication dependencies.

The main benefit of architecture-independence is the support for a high-level programming model that increases programmer productivity, paired with a self-optimising RTS that is flexible enough to adapt to different target platforms to potentially achieve performance portability. This latter aspect is challenging due to

high architectural diversity and a large number of system-level and application-level parameters that influence both performance and scalability.

3.2 RTS Components

This section explains the logical structure of the distributed GUM (Graph reduction on a Unified machine Model) RTS that implements the virtual shared memory model by discussing the main components and the key concepts behind these components.

GUM implements Glasgow parallel Haskell (GpH) and manages potentially parallel execution of GpH programs on both shared-memory and distributed-memory platforms in accordance with the recently defined PAEAN framework for shared-nothing parallelism [30]. The core of the RTS is a graph reduction engine that is written in C and implements a variant of the *Spineless Tagless G-Machine* (STG) [195].

Figure 3.2 provides an overview of the compilation pipeline. The user writes parallel Haskell code and uses libraries such as Evaluation Strategies which are then compiled using an optimising Haskell compiler (in our case the Glorious Glasgow Haskell Compiler (GHC)) [170] and the system's C compiler (in our case from the GNU Compiler Collection (GCC)).

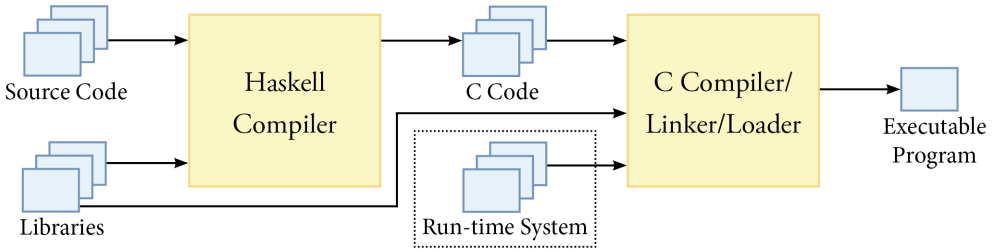


Figure 3.2: An Overview of the GpH Compilation Pipeline

First, Haskell application and library code is translated into a statically-typed intermediate language called *Haskell Core*. Core is a smaller functional language with support for variables, data constructors, literals, value and type abstraction and application, as well as `let` and `case` expressions, patterns, casts and coercions [170]. Core code is then optimised and translated to `C--` portable assembler [202], which can interoperate with C. The resulting C code is then linked with the RTS and

other C libraries to produce the actual executable, depending on whether static or dynamic linking is used. Examples of libraries include the GNU Multiple Precision Arithmetic Library (GMP) used to implement Haskell's arbitrary length `Integer`s, and communication libraries such as Message Passing Interface (MPI) or Parallel Virtual Machine (PVM). As an alternative to the `C--` compilation route, GHC more recently also supports compilation via LLVM [148, 230].

The RTS includes the following key logical components [30]:

- an *execution engine* based on graph reduction that evaluates expressions,
- thread and spark management for efficient *control of parallelism*, in particular the *scheduler* responsible for local execution and calling other components and driving load balancing, on which this work is focused,
- *memory management* responsible for virtual shared memory, implicit synchronisation, and local as well as distributed GC,
- *communications* that implicitly handles messaging and packing/unpacking of the sub-graph sent.

In addition to the four logically well-separated PAEAN components, we identify *monitoring* and *profiling* as a logically distinct component that facilitates the investigation of effects of system parameters such as the number and granularity of threads on performance, even though in practice many calls to the component are interspersed across other components.

Further components responsible for OS interaction, signal handling, exceptions, concurrency, foreign function interface, and I/O [200, 197, 169] covered through specific libraries are not further discussed here, since they are orthogonal to the purposes of the present work and hence remain unchanged. The changes to the RTS necessary to implement new primitive operations and to extend the default policies are discussed in a related technical report [22] and in Chapters 5 and 6 of this thesis.

3.2.1 Thread Management

A key design concept in GUM, as well as in the threaded, shared-memory RTS for GHC, is that of light-weight threads. For scalability and to reduce thread management overheads, light-weight threads are mapped to relatively few heavy-weight OS threads (usually one per core) in *Many-to-Many* fashion [221]. This is similar to Green Threads[4], which are managed in user space instead of kernel space, and to Qthreads [255] which offer an API for implementing light-weight threads.

In GUM, each RTS instance maintains a local thread pool for runnable threads, a spark pool for potential parallelism represented as pointers to graph structures, and blocked queues for threads waiting on a result of evaluation performed by another potentially remote thread. This improves scalability and facilitates the separation between the actual and the potential parallelism. In particular, threads can be re-used rather than destroyed and re-allocated. Parallelism is exploited over pure functions and I/O is handled by a separate thread.

Light-weight Threads Each thread is represented using a heap-allocated Thread State Object (TSO) that contains slots for register values, a stack, and some other book-keeping information, as well as a pointer to the code to evaluate the sub-expression. As heap objects, TSOs can be GC'd once they are no longer needed, but some are kept in a free-list to avoid any reallocation overhead. Figure 3.3 shows the thread state transition model.

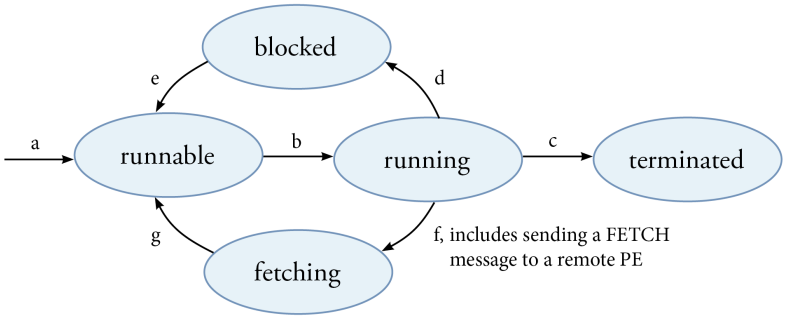


Figure 3.3: Thread States and Transitions

Initially, threads are created in the `runnable` state and added to the thread pool

[4]the name referes to the original Java threads implementation by the Green Team

via transition `a`.

A thread is `running` once it has been selected by the scheduler and passed to the evaluation engine (`b`).

A running thread then either (`c`) terminates, (`d`) becomes blocked on a closure being evaluated by another local thread (`blocking` state), or (`f`) blocks while waiting for remote thread on another PE to respond to the sent `FETCH` message (`fetching` state). Closure layout is described in Section 3.3.3.

A blocked thread will eventually unblock (`e`), similarly to a fetching thread that will eventually receive the data it needs to proceed with the evaluation (`g`).

The scheduling mechanism is discussed in more detail in Section 3.3.1, whilst the work stealing (*fishing*) mechanism for load balancing is illustrated in Section 3.3.4.

3.2.2 Communication Management

The key design principle behind GUM's communication sub-system is *latency hiding*. Because the communication latency is high in distributed architectures, computation and communication can be overlapped to hide latency. In particular, when a thread is blocked waiting on a message to arrive, another runnable thread can be scheduled to run instead, as discussed in detail in Section 3.3.1. This follows the *evaluate-and-die* execution model [65, 198, 195].

In the beginning of an execution, a manager process spawns a specified number of GUM instances, usually one per core. We will use the term PE to include the RTS instance. These are initialised and then one PE is elected to be the main PE that will evaluate the `Main.main` closure that acts as the main entry point for the application, whilst others will start off fishing for work.

A program terminates once the main thread has finished evaluation or if an error occurs, by the main PE sending a `FINISH` message to all RTS instances.

GUM uses a broadcast and a barrier during initialisation and termination, and point-to-point communication during execution. Currently there is no support for fault tolerance, relying on low-level mechanisms provided by communication libraries and the OS. For instance, the underlying TCP/IP protocol guarantees the arrival

of GUM messages.

The communication is *asynchronous* and implemented using layers. A high-level API is used in most RTS modules and can be implemented using a low-level library such as MPI or PVM with support for send/receive communication functions. The required changes are localised in only 3 out of around 150 RTS modules [235].

If no local work is available, the RTS instance will attempt to obtain remote work as discussed in Section 3.3.4.

Fetching and Implicit Synchronisation Another important design principle is *implicit synchronisation* that hides the intricacies of synchronisation as well as communication from the programmer, facilitating the expression of parallelism, whilst delegating performance tuning mostly to the RTS. The benefit is worthwhile: race conditions and deadlocks are ruled out by design. The former are avoided by using private memories only accessible by the owner PEs, whilst sharing requires communication. The latter are avoided through the structured use of communication primitives inside the communication layer, because synchronisation is mediated through the graph being reduced.

Moreover, synchronisation on graph nodes can be used to implement sharing and call-by-need semantics [253, 195] and to avoid potential duplication of work.

If a thread requires a part of the graph, it will be fetched from the remote PE rather than evaluating it directly, as shown in Figure 3.4. A global address (GA), which is a unique identifier across the physically distributed heaps, is used to refer to the sub-graph[5].

However, if the graph node is local, and not under evaluation, the thread will start to evaluate it, and will overwrite the graph node representing a yet unevaluated expression by a Blackhole (BH) which acts as an implicit synchronisation point. Thus, if another thread attempts to evaluate an expression that is currently under evaluation by a local thread it will enter the BH, which will result in it blocking and

[5]we use the standard UML Message Sequence Charts graphical notation to illustrate the messaging protocol and relevant message types: top boxes represent independent RTS instances and arrows labelled with message types and key payload contents denote messages being exchanged in the order from top to bottom; Haskell syntax is used to denote a list of pairs of GAs (`ACK` payload)

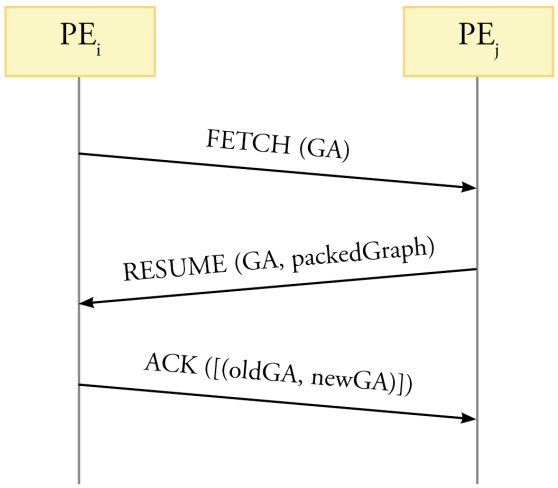


Figure 3.4: Fetching Protocol

adding itself to the associated waiting queue, returning the control to the scheduler. Once the evaluation is complete, the BH is overwritten by the result value and all awaiting threads are notified and moved to the runnable pool. The use of GAs to implement sharing across private heaps is discussed in Section 3.2.3.

Message Types and Format Messages in GUM comprise a header and a payload. Message types are specified in the header denoting the payload contents to expect and are used to determine the way to handle each message and to respond according to the protocol. Table 3.1 summarises GUM's message types.

The message header also contains the source PE and the destination PE which enables the main GUM communication protocol to remain stateless. The first six message types reflect the main protocol, whilst the remaining three are used during the start-up and termination phases. The payload part of the message carries its own meta information depending on the message type.

For example, each `SCHEDULE` message specifies the buffer size required to be able to unpack the graph packed in the payload of the message. Some messages, such as `FISH`es, are of small fixed size and offsets are used to extract particular values.

Table 3.1: GUM's Message Types

Type	Description
FISH	request for work (steal attempt; original, forwarded, or expired on the way back to origin)
SCHEDULE	response with some work (spark(s) and nearby graph)
FETCH	request for required remote data
RESUME	response with some graph data
ACK	positive acknowledgment of reception of required sub-graph and indirection information update
NACK	negative acknowledgment, indicates failure to receive the required sub-graph
PETIDS	synchronisation request, sent at initialisation by the main PE
READY	synchronisation response from worker PEs to main PE
FINISH	termination request from main PE, sent to other PEs at the end of execution

3.2.3 Memory Management

GUM implements GpH by supporting distributed graph reduction. Each graph node represents computation that is potentially shared among multiple PEs that require its result. Thus, there are the independent PE-local heaps for local reduction alongside the *virtual shared memory* overlay that holds the shared graph. Once a node has been evaluated it is replaced by the result, which is in turn sent to all the PEs that require it. GUM's design, based on private heaps with some potential sharing across them, is scalable as most garbage collection (GC) [135] can be performed *locally* without the need for communication and synchronisation. In particular, there are two different GC layers: local GC is independent and global GC which only applies to a small subset of graph nodes.

Local Generational Garbage Collection GUM uses a *generational* garbage collector that is either *copying* or *compacting* depending on the RTS flags set, thus avoiding using a *stop-the-world* design which has significant scalability limitations. Heap objects that survive for a long time are promoted from the initial and frequently GC'd heap area (called *nursery*) to a different heap region that is GC'd less often. This GC scheme assumes that most heap objects will expire after a short period of

time allowing the associated memory to be reclaimed, whilst older objects are likely to continue being needed.

Distributed Weighted Reference Counting Usually, only a small subset of the graph is shared across PEs, which is collected using distributed GC based on *weighted reference counting* [31]. The *Global Indirection Table* (GIT) that maps global addresses (GAs) to local addresses (LAs) and vice versa is also used as a source of roots for GC.

Each GA represents an outgoing pointer and has an associated PE identifier, a local index, and a weight, which represents the percentage of all references to this object. GAs are used to link objects across PEs, whilst LAs are the local addresses (pointers to heap) that may have many GAs. If a closure is shared across PEs, its initially assigned weight is evenly split between the local and the remote PEs that maintain a reference to it. The underlying invariant is that the sum of weights for each GA in GIT for all out-pointers and the weight in the owner's GIT plus the sum of the weights in all messages in transit equals a fixed maximum weight at all times.

The GIT table ensures the mapping is up-to-date and is rebuilt during GC. The mapping from GAs to LAs facilitates finding of the correct LA when processing incoming `FETCH` messages. The reverse mapping is used to identify whether a heap object that is being packed has already been packed to maintain sharing. Moreover, the mapping needs to be updated if the location of the object changes after the transfer of a graph structure, which may require getting a new GA and replacing the old GA with an indirection heap object. Local GC also updates the GIT as LAs may change, which is the source of additional overhead.

Once the owner has the full weight back, the memory can be released as it is no longer required. An example of using a GA is discussed in Section 3.3.3.

3.2.4 Workload Management

Work needs to be distributed across PEs to enable effective exploitation of parallelism. The aim is to maximise utilisation of the available PEs whilst avoiding communication overhead and achieving the performance goal. In this thesis, the

focus is on decreasing execution time and increasing scalability.

Work can be either *actively* (eagerly) off-loaded to other PEs (work pushing) or *passively* (lazily) obtained by idle PEs who ask other PEs for work rather than waiting for work to arrive (work stealing; cf Section 3.3.4). GUM's default load balancing mechanism is random work stealing, where victims are chosen at random by idle PEs. GHC-SMP supports work pushing through allowing direct access to other PEs' spark pools.

3.3 Policies and Mechanisms

We follow the system design principle of separation of concerns between *policies* and *mechanisms* [221]. Policies are focused on what is to be achieved, which can be formulated as a plan or a set of rules or the requirements and conditions that lead to a desirable outcome [42, 224]. In contrast, mechanisms describe how the policies are to be supported or enforced to achieve the set goal and allow for significant freedom in the choice of implementation techniques that include selection of algorithms and data structures. At a lower level, the implementation is concerned with the specific choice of suitable algorithms and data structures using a particular programming language.

This separation is beneficial for it gives more flexibility to extend the system through a set of localised changes, allowing a policy to be exchanged without the need to change the mechanism. Therefore, a mechanism should avoid unnecessarily restricting the choice of policies, and vice versa, the policy should not presuppose a particular mechanism or implementation.

For example a load balancing policy could specify the goal to minimise run time whilst maintaining high average utilisation across PEs, where work pushing or work stealing could be used as a mechanism to enforce the policy. Assuming we have chosen a work stealing mechanism, we can tune its components such as victim selection and selection of sparks to donate. Finally, the implementation of the mechanism offers further design decisions: for instance, work stealing could use a centralised or a decentralised approach, whilst a spark pool could be implemented

as a priority queue or as a lock-free dequeue. Below we describe GUM's policies along with the corresponding mechanisms and their implementation in more detail.

3.3.1 Scheduling

The scheduler is the central RTS component and is responsible for the following tasks, delegating some of them to the respective RTS sub-components:

1. perform garbage collection, if necessary;
2. process incoming messages, if any have arrived;
3. run a thread, if there is at least one runnable thread available;
4. or if possible, activate a local spark;
5. otherwise, look for a remote spark (attempt to steal work from other PEs).

Listing 3.3 illustrates GUM's scheduler (cf `rts/Schedule.c`). After 1) performing GC, if necessary, and 2) processing any incoming messages, e.g. notifying blocked threads once the values they have been waiting for have become available, 3) the *unfair* scheduler selects a runnable thread to run in *First-Come-First-Serve* (FCFS) fashion. The chosen thread then *non-preemptively* either runs to completion, or until the space is exhausted, or it blocks on a shared computation under evaluation by another thread or on remote data access, or it is terminated due to an error. From a design point of view, an unfair scheduler improves time and space behaviour by avoiding book-keeping and context switching among light-weight threads, but risks starvation and makes speculation and concurrency more difficult to exploit [237].

If no runnable thread is available, the scheduler will 4) look for work in the local spark pool. If there are sparks, the oldest will be turned into a thread and evaluated (FIFO). Otherwise, if no potential work is available locally, the scheduler will 5) attempt to *steal* a spark from a randomly chosen remote PE (cf Section 3.3.4).

When a thread attempts to evaluate a sub-graph, the sub-graph may either be under evaluation or unevaluated. In the latter case the thread can evaluate it and mark it as being under evaluation, effectively *subsuming* the corresponding spark.

In the former case, if the sub-graph is either under local or under remote evaluation, the thread will block and wait for the local thread to update the root, or for a message to arrive with the required data, respectively.

```
while (not terminated) {   // core of the schedule() function
  if (needGC())
      performGC()   // 1. reclaim unused memory, if necessary

  if (incomingMessages())
      processMessages()    // 2. process any incoming messages

  if ((t = findRunnableThread()))
      run(t)         // 3. run a thread, if there is a runnable one
  else {
    if ((s = findSpark()))
        activateSpark(s)   // 4. else, use a spark, if available
    else
        getRemoteSpark()   // 5. otherwise look for remote work
  }
}
```

Listing 3.3: GUM Core Scheduler Loop (Pseudocode)

Sparks for Advisory Parallelism Sparks are kept in a separate local pool on each PE. Sparking is cheap, as it adds a pointer to a closure heap object representing the expression to be evaluated (a thunk) to the spark pool. Note that STG uses programmed graph reduction and therefore no explicit graph is maintained in the heap, but rather TSOs, sparks and objects representing closures.

Figure 3.5 depicts potential states of a spark: once created a spark representing potential parallel work may either be *converted* into a thread to be evaluated in parallel or it may end up not being converted. Here, a spark may be already evaluated (i.e. pointing to WHNF) as in the *dud/fizzled* case, *discarded* in the case that the spark pool is already full, or *garbage-collected* if it was not needed, i.e. it was not shared with the rest of the computation. It is a separate question as to whether the work represented by a spark was useful work.

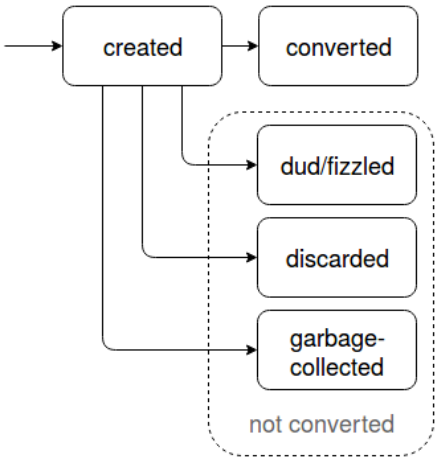


Figure 3.5: Spark States

The pool is implemented using an efficient lock-free dequeue [55] which allows the owner, i.e. the PE that created the dequeue, to use one end locally for popping and pushing (LIFO), whilst older sparks are stolen off the other end using a single atomic compare-and-swap operation (FIFO). This choice is based on the insight that older sparks are often associated with computations of larger granularity (similar to the Breadth-first Until Saturation then Depth-first (BUSD) mechanism [49]). Note that the overhead is absent unless two threads actually attempt to dequeue the same spark. Sparks are discarded if they have been already evaluated, e.g. through thread subsumption, or if the spark pool is full. This can influence the actual degree of parallelism and thread granularity at run time.

3.3.2 Granularity Control

Control of granularity, i.e. the computational size of tasks, is crucial for ensuring good performance, since it addresses balancing granularity and actual parallelism [162, 109]. As parallelism is very fine-grained in graph reduction, because every sub-expression can be evaluated in parallel, the overhead of creating new threads may overwhelm the benefit of evaluating many relatively small computations in parallel. Hence, there is often a need to throttle the available parallelism and keep actual parallelism at a fraction of the available.

Additionally, it is critical to ensure that sparks associated with larger granularity are kept for parallel execution and export, whilst sparks that are associated with too fine-grained computations are discarded. Moreover, granularity control interacts with load balancing, since having too little actual parallelism may lead to load imbalance, as can a large variation in granularity. For example, if a very large computation is evaluated last, all other PEs would have to wait for the straggler.

One RTS-level mechanism for self-throttling granularity control is *thread subsumption* through inlining of the child spark into the parent thread that requires its result. This effectively increases GUM's architecture independence by adaptively throttling granularity depending on the number of idle PEs, using the mechanism as described in Section 3.3.1. In particular, D&C computations are suitable for thread subsumption due to their tree-like computational structure.

Although not the focus of this work, it shoud be noted that the user can improve granularity by using application-level techniques such as thresholding, chunking, and clustering, which complement the RTS-level mechanisms. Essentially, these techniques logically increase the nesting of the data structures and exploit parallelism across whole groups of elements instead of working on each single element in parallel. This leads to reduced parallelism, i.e. it reduces the number of created sparks and consequently of threads. This results in increased granularity, as on average threads are now associated with larger computations. Overall, granularity control is challenging due to the difficulty of reliably predicting spark sizes in advance, either statically or dynamically, and hence heuristics are often used in practice.

3.3.3 Data Locality

Data locality in the context of distributed graph reduction refers to keeping data required for computations local or nearby rather than to cache-related behaviour. We will use the size of the global indirection table, the format of which we described in Section 3.2.3, as a means-based metric for *fragmentation* of the global heap, as each GA entry in the table represents a cross-PE pointer referring to exactly one shared closure. Hence, the larger the table size, the higher the degree of graph sharing

across PEs. This is relevant, as inter-PE sharing requires additional communication, which contributes to the overall overhead and may not always be overlapped with computation. Although the number of GAs is only an approximation of the absolute shared heap size, it is sufficient to allow relative decisions.

Heap Organisation Closures and other objects, such as TSOs that represent light-weight threads and state information, are allocated from the heap and are automatically reclaimed by the GC once no longer needed. The advantage of this scheme is that both internal objects and different types of closures can be handled uniformly by the garbage collector [195]. Figure 3.6 illustrates the generic layout of heap-allocated objects in the RTS.

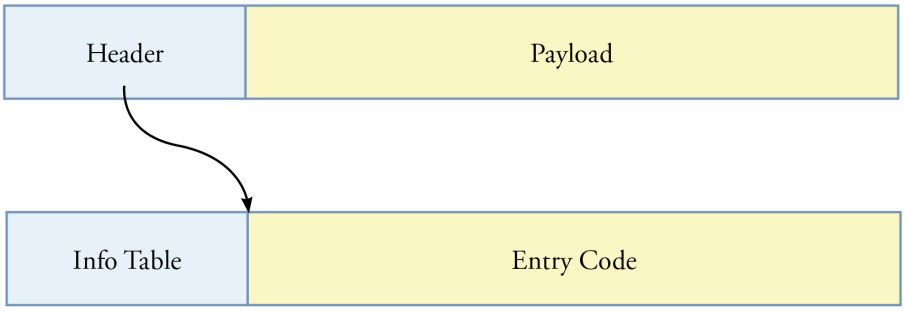


Figure 3.6: Generic Heap Object Layout

A heap object consists of a header and a payload that contains pointers to other objects and non-pointer data as described by the header. The header points to the entry code used to evaluate the closure. An info table resides at an offset just before the entry code and can be examined to check the type of the object to ensure it is handled appropriately. For example, a function closure contains references to its free variables in the payload.

Virtual Shared Memory The description below is based on [237, 30]. GUM's virtual shared memory implementation allows sharing of graph nodes across PEs by globalising closures, i.e. by assigning a GA to each thunk, instead of recording

a GA inside of each closure. By contrast, data in normal form (NF) is copied across nodes [158]. Each PE uses a Global Indirection Table (GIT) to maintain the mapping between local and global addresses. The size of the transmitted sub-graph is limited by the fixed upper limit for the packet size, which can be set using an RTS option. If the graph does not fit into the packet indirection closures are packed and transmitted using multiple packets instead.

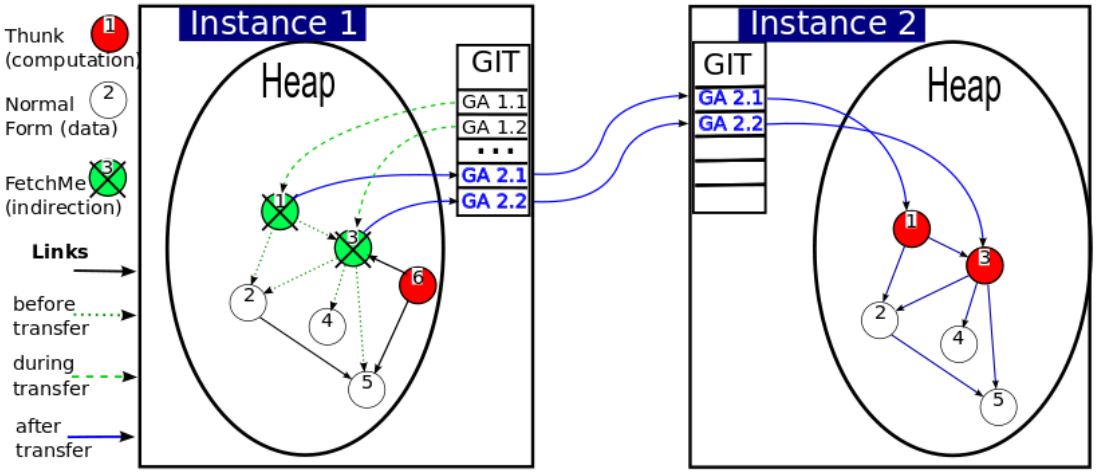


Figure 3.7: GUM's Virtual Shared Heap (from [30])

Figure 3.7 shows how parts of the graph are shared across PEs. Nodes 1 and 3 on PE1 (initially referred to by using temporary GAs `GA 1.1` and `GA 1.2`), are indirection closures (`FetchMe` objects) which after transfer refer to remote sub-graph using GAs (`GA 2.1` and `GA 2.2` on PE2). Nodes number 2, 4, and 5 are copied across as they are data in NF. To evaluate node 6, PE1 will use the fetching protocol to obtain the necessary values from PE2. The GIT needs to be updated after each GC to maintain correct mapping, which contributes to the overhead and is the reason for keeping the number of GAs as low as possible.

This mechanism is based on the assumption that most of the graph nodes remain unshared and there is no need to globalise them [199]. Moreover, GUM's distributed GC is currently unable to collect cycles across PEs, which remain in the GIT until the end of the run[6]. However, suitable schemes exist [31] to handle this uncommon

[6]communicating values and thunks that are part of a cycle requires no special treatment

case and could be implemented in the future.

Graph Packing A shared sub-graph needs to be serialised before it is sent to remote PEs over the network. The packing proceeds in breadth-first order to allow local cycles to be reconstructed when unpacking the message using the implicit ordering. Additionally, the packet header specifies the size of the pack buffer needed to unpack the graph.

The receiving PE checks whether a more defined copy of the sub-graph is available locally and if so uses it, whilst updating the GIT. Some nearby graph that is likely to be needed is usually included the message, which is similar to pre-fetching, resulting in data being packed somewhat eagerly.

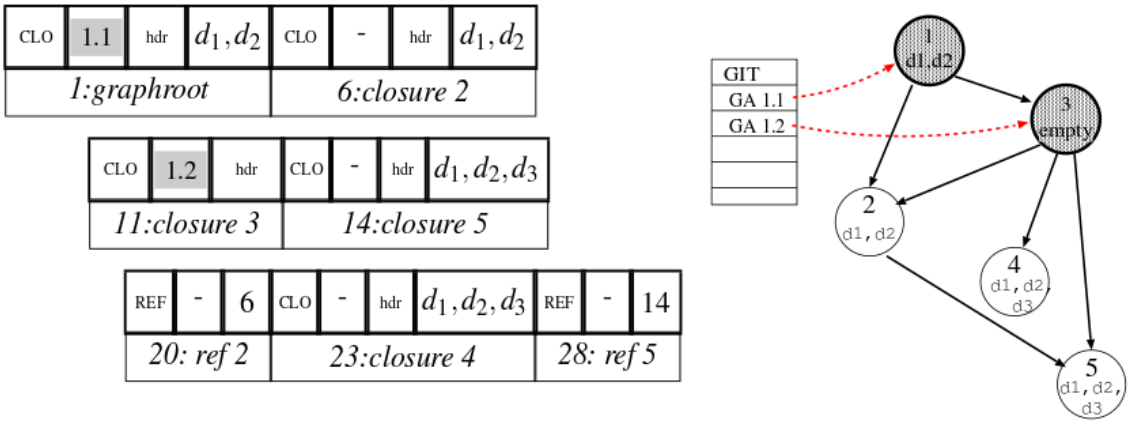


Figure 3.8: GUM's Graph Packing (from [30])

Figure 3.8 depicts a packed graph including closures, that are identified by a tag and a GA, and references to shared graph nodes. For example on the right-hand side, node 5 is shared by nodes 2 and 3 and hence is packed only once and then referenced. The graph root is found at the offset of one and subsequent closures and references at their respective offsets, which depend on the cumulative size of the preceding ones.

3.3.4 Load Balancing

In GUM, the load balancing policy that involves distribution of work across PEs is realised using a work stealing mechanism (also called *fishing*) and aims at reducing the overall idle time across PEs, whilst keeping the communication overhead as low as possible and ensuring the highest possible performance.

Work Stealing Work stealing is a *passive*, i.e. receiver-initiated, decentralised workload distribution mechanism used in many parallel language run-time systems [38, 85, 67].

The two main decision points are:

- *where to steal from*: victim selection by a thief or selection of forwarding destination by victim with no sparks available for export; function `choosePE()`.
- *which spark to export*: decision made by a victim that has exportable sparks; function `findSpark()`.

These decisions are the main points where we can intervene by letting extended mechanisms take decisions differently from the baseline system. Listings 3.4-3.6 show the work stealing pseudocode, whilst Figures 3.9-3.11 illustrate the message types and the protocol. Refer to Table 3.1 for message type descriptions.

As shown in Listing 3.4, the `choosePE()` function is used by the default mechanism to select a victim at random and send out a `FISH` message, as long as the maximum number of `FISH` messages in transit was not exceeded and delay between sending consecutive `FISH` messages is adhered to.

By default, a victim that receives a `FISH`, selects the oldest spark for donation and sends it back to the origin PE (see Figures 3.10 and 3.11). The thief acknowledges reception of a spark by sending an `ACK` message with an updated list of pairs of old GAs and new GAs to the victim (Listing 3.6).

```
1 getRemoteSpark() {                          // thief looking for work
2   // ...
3   if (outstanding_fishes < MAX_FISHES)
4       if (next_fish_to_send_at <= now()) // FISH delay has passed
```

```
            sendFish(to = choosePE() , origin = thisPE, age = 0)
}
```

Listing 3.4: GUM Work Stealing: Thief Sending a `FISH`

The tunable fish delay and delay factor determine the pause between sending consecutive `FISH` messages to avoid swamping the network with messages in addition to a limitation on the number of outstanding fishes (currently one per PE by default). The variable `outstanding_fishes` is updated when sending or receiving a `FISH`, whereas the delay and delay factor are set at RTS startup, either to a default value or to a value of the corresponding RTS flag, and delay is multiplied by the factor every time an expired `FISH` message returns to the originating PE.

```
processMessages() {                 // victim's response to a FISH
  //...                 // msg.type == FISH and msg.origin != thisPE
    if (msg.age == MAX_AGE)     //   return expired FISH to origin
        sendFish(msg, to = msg.origin)
    else if (s = findSpark())  // export a spark if available
        sendSchedule(pack(myPEid, s), to = msg.origin)
    else {
        msg.age = msg.age + 1
        sendFish(msg, choosePE()) // forward FISH to another PE
    }
  // ...
}
```

Listing 3.5: GUM Work Stealing: Victim's Response

```
processMessages() {
  // ...
  if (msg.type == SCHEDULE) {      // THIEF either receives work ...
      s = unpack(msg)
      add(spark_pool, s)
      updateGIT()                  // send updated GAs to owner
      sendAck([(oldGA,newGa)], to=msg.sender)
  }
  // ...                           // ... or expired own FISH
  if (msg.type == FISH && msg.origin == thisPE) {
```

```
11        outstanding_fishes--;
12        last_fish_arrived_at = now()
13        fish_delay = fish_delay * fish_delay_factor
14        next_fish_to_send_at = last_fish_arrived_at + fish_delay
15    }
16    // ...
17 }
```

Listing 3.6: GUM Work Stealing: Thief Handling a Response

Single-Hop Successful Fishing Attempt: Figure 3.9 demonstrates the case where the thief got lucky by randomly selecting a victim that was able to donate a spark. After the victim is selected, a `FISH` is sent and the thief proceeds with the scheduling loop. Once a victim has received the `FISH`, it selects a spark to donate, packs it with a nearby sub-graph of tunable size that is likely to be needed and sends it in a `SCHEDULE` message back to the thief. The thief responds with an `ACK` message that is used to update global addresses in the recipient's GIT (see Figures 3.9 and 3.10) that have changed as a result of spark movement.

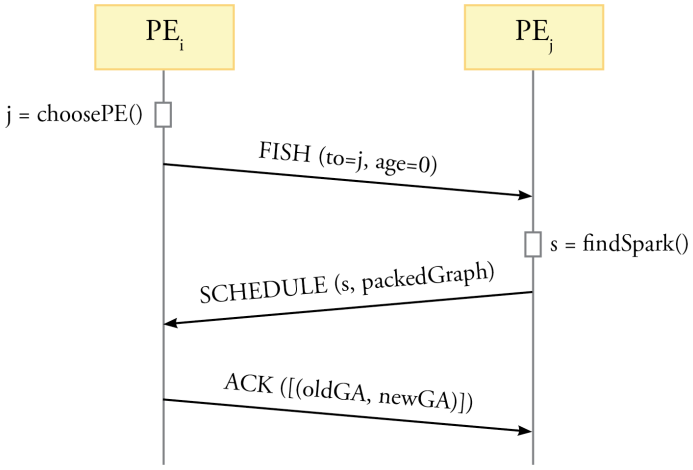


Figure 3.9: Single-Hop Successful Fishing Attempt

Multi-Hop Successful Fishing Attempt: Often a `FISH` has to travel over multiple hops to find a victim that can donate a spark, as shown in Figure 3.10.

The protocol starts as for the single-hop case, but deviates when a victim receiving the `FISH` has no sparks and forwards the request to another randomly chosen

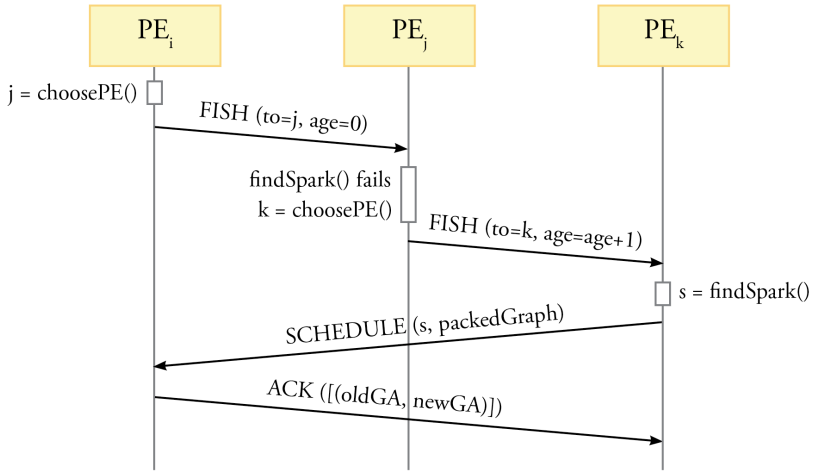


Figure 3.10: Multi-Hop Successful Fishing Attempt

PE (see also Listing 3.5). Once a suitable spark is found the protocol ends with the exchange of `SCHEDULE` and `ACK` messages as discussed above. Note that the `FISH` contains the PEid of the thief, so that the spark can be sent directly to the thief.

Unsuccessful Fishing Attempt: Every time the `FISH` is forwarded its age is incremented and it can expire if it reaches the maximum allowed age. Figure 3.11 presents the protocol in this case, where the expired `FISH` is sent back to the thief, which then may send out a new `FISH` after a short delay (see also Listing 3.6).

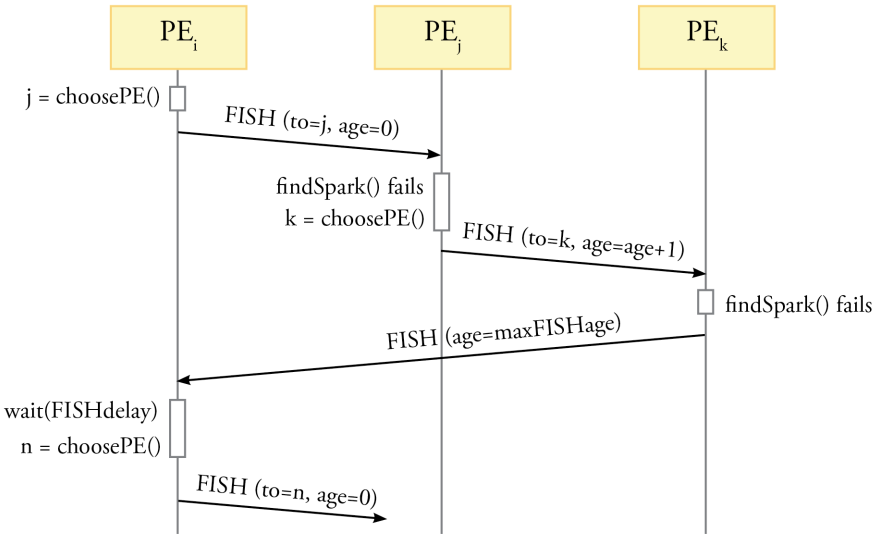


Figure 3.11: Unsuccessful Fishing Attempt

Although useful for avoiding scheduling accidents when one PE turns most of

the sparks into threads and others stay idle, *thread migration* [77], where a thread can be shipped to a remote PE, is currently not supported. Such accidents are deemed unlikely, as in practice parallelism is usually rather fine-grained in functional programs and the cost of migrating a thread often outweighs the benefits.

3.4 Adaptivity

Adaptivity is a key feature of systems that are capable of coping with dynamically changing circumstances such as load variations. Architecture-independence at application level requires some degree of architecture-awareness and adaptation at implementation level to achieve high performance across different architectures. Adaptivity is enabled by using a *feedback loop* at run-time that allows the system to monitor itself and its environment and to tune the employed mechanisms based on that information, as illustrated in Figure 3.12 depicting GUM's control model. This is more flexible than a tuning cycle involving a human expert manually tuning the parameters after examining the profiling data and mostly requiring interruption of the current application run or even recompilation. Manual tuning appears increasingly less feasible in practice due to rapidly increasing software complexity, larger parameter spaces and an expanding architectural landscape.

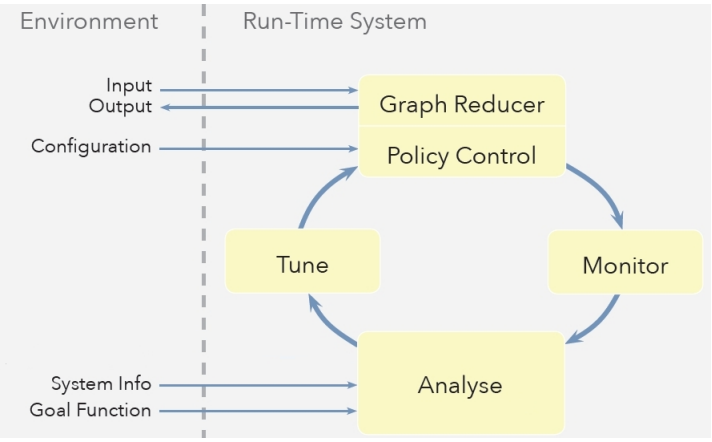


Figure 3.12: GUM's Control Model

Profiling has always played a key role in performance evaluation of computer systems [130]. It is complemented by analytical techniques, which are however of limited applicability and accuracy, as abstraction often requires severe simplifications to fit

existing formalisms that often are inadequate in reflecting the diversity and complexity of systems and applications operating in dynamic environments. Simulation, which covers the middle ground and allows exploration of a larger design space, is somewhat less abstract and idealised and sometime can be parameterised by actual target platform characteristics.

3.4.1 Monitoring and Tuning Classification

Table 3.2 illustrates the temporal relationships between monitoring and tuning based on the control loop model of adaptation, with a focus on classifying the degree of adaptivity of a given mechanism or system.

- When and how often is monitoring performed?
- When and how often is tuning performed?

We distinguish between several points in the design space based on these questions, and focus on hard-coded control (never monitor, never tune), configuration (never monitor, tune at start-up time), and dynamic adaptation (monitor often, tune often), as these appear most relevant for language RTS design, highlighted in Table 3.2.

Table 3.2: Temporal Relation between Monitoring and Tuning

		Tune		
		never	once	often
Monitor	never	**hard-coded control**	**static at start-up (configuration)**	random mutation
	once	detached trigger	snapshot at run time (one sample)	triggered mutation
	often	detached sensor	decision at run time (multiple samples)	**dynamic (event-based)**

The least general category, and our baseline case, is *hard-coded control*, which is very common in software systems and lacks flexibility. Often manual code adaptations are required to respond to change and in most cases this is prohibited at

run-time. However, adaptation is still possible using hard-coded control if a decentralised algorithm is employed that embodies a capability to react to environmental and system state changes.

For instance, the work stealing algorithm for load balancing and work distribution may hard-code victim choice and spark selection for export, but is still classified as adaptive because it reacts to load changes. Nevertheless, the decisions are made by the system programmers based on their domain knowledge and experience and are fixed at compile-time. Changing the hard-coded parameters requires recompilation and restart of the application.

The least useful categories include isolated components such as *detached trigger* (monitor once, never tune) and *detached sensor* (monitor often, never tune) and are briefly mentioned for completeness. The usefulness is limited as no action is taken, however, if a human observer is involved, the output may still be used in a manual fashion, e.g. by analysing the logs produced by the sensors.

A dual class of systems we will not further discuss includes *random mutation* (never monitor, tune often), which may have some merit in simulations or generating random test input, but in adaptive systems would lead to instabilities, as the state of the system and the behaviour could change regardless of the need, resulting in sub-optimal trajectory. Similarly, *triggered mutation* (monitor once, tune often) can not ensure that multiple tuning actions reflect the actual need.

Another point in the design space similar to hard-coded control is *static configuration* at start-up time, since the number of configuration options is usually hard-coded. The benefit of the scheme is that it requires no re-compilation and exchanges the configuration settings via a config file or some RTS parameters. Some systems allow changes to the configuration at run-time and hence enable re-configuration. These can rather be considered dynamic and potentially adaptive. However, configuration relies on the a-priori knowledge of the entity which sets the configuration parameters and not on monitoring.

A related design point is the *snapshot* (monitor once, tune once), where the tuning decision is made based on a single observation, which may be inaccurate,

unless the sample is representative of the behaviour of the system. For instance, a program can query the operating system to check for availability of accelerator hardware and decide to off-load parts of the computation to improve performance. Additionally, it is difficult to determine the best time to make such an observation.

The *decision* category (monitor often, tune once) is based on multiple samples and may involve some statistical analysis to determine the following tuning action. The key limitation leading to inflexibility is that the tuning step is performed only once. An example is an emergency sub-system that constantly monitors system state and acts appropriately (e.g. gracefully shuts down the system) once the emergency state is detected. Such a system could be used to support fault tolerance by invoking a specific fault handling component.

Finally, the most general and flexible option is *dynamic adaptation* at run-time (monitor often, tune often), where multiple decisions and actions are taken based on many samples. This option can emulate any other option depending on the possible use of the monitoring information (e.g. ignore or use once) and tuning activities (e.g. decide never to tune, which is equivalent to tuning often to the same parameter values as before, or tune once).

A further sub-division can be made based on the frequency of monitoring and tuning: periodic or event-based. Event-based is a more general case, because the periodic case is equivalent to the event-based with fixed intervals between events.

The benefit of using a dynamic mechanism may be offset by the cumulative profiling and tuning overheads and hence constitutes a trade-off between the frequency of monitoring and tuning and the level of responsiveness of the system.

3.4.2 Parameter Selection

The choice of suitable parameters and values poses yet another challenge. Monitored system parameters include total counts of different messages and threads in different states, providing an overview of the system, as well as per-thread profiles for more fine-grained control. Table 3.3 summarises key parameters available in GUM.

For example, a high overall number of messages (`FISH`, `FETCH`), large indirection

Table 3.3: A Selection of Observable and Tunable Parameters

high-level concept	Monitoring		Tuning		
	start-up	run time	compile time	start-up	run time
load balance and locality	latency, nodes, PEs	per-thread info, table size, load	max FISHes sched algo	location of main PE	victim and spark choice
parallelism, granularity	nodes, PEs, code/object	per-thread info, pool/queue size	(as for LB)	(as for LB)	inlining policy
communication	latency, bandwidth	avg packet size, rate of transm.	packet size, max closures	nodes, PEs, main PE loc	max FISHes max closures

table size, and many blocking threads hint at potential load imbalance. The challenge is to devise general ways to respond to such situations. Further parameters include architectural information such as the number and computational power of PEs and characteristics of the memory and network hierarchy, which can help determine where to look for work or to which PE to off-load work to maximise the benefits and reduce the costs. Empirical evaluation is necessary to assess tuning effects and to justify the selection and tuning of the chosen parameters.

3.4.3 Tuning GUM

Here we briefly discuss sources of adaptivity in GUM. The main adaptive mechanisms are work stealing for load balancing and thread subsumption for control of granularity and parallelism.

Thread subsumption is an adaptive mechanism as it reacts to load in relation to available PEs. As discussed in Section 3.3.1, thread granularity is indirectly increased based on load, by inlining child sparks into a parent thread, thus effectively reducing the actual degree of parallelism and increasing average thread granularity. This is useful if the system is saturated with sparks and under relatively heavy load, for it avoids thread creation overheads and improves locality, as the parent thread is likely to require the results of a computation associated with its child sparks.

Work stealing is another example of an adaptive mechanism. Even in its baseline version the mechanism adapts to changing load patterns and balances the load across the system. According to the suggested classification, the algorithm is dynamic as it

uses events (running out of work) to generate work request that are sent by the idle PEs, resulting in rebalancing of the load, when PEs that have work react to further events (receipt of the `FISH` messages) by sending sparks to idle PEs, thus distributing the work. In particular, the set of local decisions results in global behaviour change. Additionally, work stealing utilises an adaptive back-off mechanism by tuning a delay factor to avoid swamping a lightly loaded cluster with stealing requests based on recent fishing failures.

We argue that adaptivity is not a binary concept, as it is possible to recognise different levels of adaptivity, for instance direct and indirect. Moreover, adaptivity appears crucial for implementing support for performance portability as the a-priori knowledge available at compile time and at start-up is fundamentally limited and performance in many cases depends on parameters only known at run time. Examples of such parameters include task granularity, dynamic degree of parallelism, stream data size, shape or degree of sparseness.

3.5 Summary

This chapter focused on the design and implementation of the GUM RTS, illustrating architecture-transparent control of parallelism at run time on top of the virtual shared memory for seamless execution on distributed-memory architectures. The GpH language and Evaluation Strategies abstractions were introduced as the means to express parallelism and control granularity at application level. Moreover, we discussed the Graph Reduction evaluation model and the Unified Machine Model that enables architecture transparency and adaptation.

Relevant key concepts and the driving forces behind GUM's design, such as lazyness, advisory parallelism, determinism, as well as implicit synchronisation and communication with latency hiding, were described alongside the related policies and corresponding mechanisms controlled by the RTS. These mechanisms include scheduling, thread management, memory management (including GC and management of globally shared closures), load balancing using work stealing, granularity control, and data locality.

In particular, the key work stealing decisions that influence load balancing, i.e. which PE to steal from and which spark to donate, offer places to intervene and adapt the mechanism at run time. This suggests areas for investigation and extension both from the thief's and the victim's point of view. More specifically, the thief chooses the victim entirely at random, a decision that can potentially be improved by using relevant historical system-level information. Additionally, the victim donates the oldest spark in response to thief's request. We envisage that using system-level information a related spark can be donated.

Another central theme is adaptivity and the associated flexibility exemplified by thread subsumption that adaptively controls granularity and parallelism degree in GUM. To the best of our knowledge, the adaptivity classification scheme introduced in Section 3.4.1 is a novel contribution that enables classification of RTSes based on the frequency of monitoring and tuning actions. We identify the dynamic event-based approach as the most flexible due to its ability to emulate any other form of adaptation.

Based on the GUM model and key intervention points identified in this chapter, we proceed to characterise a set of parallel functional applications in the following Chapter 4 using relevant metrics such as heap residency and size of the global indirection tables as well as thread granularities and communication rate.

Chapter 4

Characterisation of Parallel Functional Applications

This chapter presents a profiling-based characterisation of eight small and medium-sized parallel functional applications with respect to the wider issues of load balancing and data locality. To gain an insight on these, we consider specific characteristics available through profiling and relevant for adaptive management of parallelism, such as the potential and actual degree of parallelism, scalability, communication degree, thread granularity, and memory usage. These means-based metrics are useful in revealing differences between applications and among run-time systems. Experiments are conducted on a modern 48-core server with a NUMA architecture and on a Beowulf-class cluster consisting of multiple 8-core nodes using up to 64 cores in total, as described in Section 4.3.1. This characterisation, paired with information on the model of GUM from the previous chapter, is the basis for the choice of heuristics presented in Chapters 5 and 6, and is an extended version of our 2015 paper [25].

4.1 Application Characterisation Studies

To provide broader context, we discuss several characterisation studies from the literature and the differences to our purposes. Application characterisation studies can be used to assess and inform the design of computer architectures, as well as run-time systems and virtual machines such as JVM [155] and CLR [179], using

well-known benchmarks, e.g. SPEC, STREAM, DaCapo [118, 175, 34], to assess processor performance and memory throughput. More relevant to our work, the GHC nofib suite [192] has been used for a long time as the standard benchmark suite in Haskell, in particular to assess the effectiveness of new compiler optimisations. Another common use case is workload characterisation, i.e. the comparison of the coverage of a parameter space by several benchmark suites or sets of workloads to assess their similarity.

The widely cited Berkeley Report [12] introduces twelve *motifs* (originally termed *dwarfs*, when there were seven) that describe common computational kernels (e.g. graph algorithms, structured grid, dense and sparse matrix operations) and reviews their use in different application domains to justify their importance. However, this view is very high-level as multiple motifs can overlap in terms of run-time behaviour and parallelism patterns used.

Another study suggests a classification based on last-level-cache (LLC) access behaviour and uses animal names [259]: Turtles do not stress LLC much because of a small working set or few memory instructions; Sheep are well-behaved and unlikely to be disturbed by others; Rabbits are sensitive to cache usage patterns; Tasmanian Devils are highly undesirable since they interact badly with almost any other program and overall system performance should benefit from their isolation.

In contrast to the above purposes, we aim to discover system parameters that could be monitored and dynamically tuned within the GUM RTS and that could suggest potentially beneficial interventions or improved heuristics. In our work, we focus on a run of a single application on a dedicated cluster and do not investigate interference patterns.

Choice of Characteristics Similar to the characterisation of the SPLASH-2 and PARSEC benchmarks [32], we use several common characteristics such as the working set size (maximum heap residency), communication-to-computation ratio and the number of light-weight threads as well as their granularity.

Specific to the implementation of a virtual shared memory abstraction, we also measure memory allocation rate and collect detailed information on threads blocking

and fetching times and counts. In particular, along with studying inter-PE sharing in general terms, we focus on the size of global address tables as a means-based metric of inter-PE sharing, as well as on the amount of graph sent to assess locality. These characteristics as important in non-strict functional setting as the graph reduction evaluation model puts additional pressure on the heap usage [169].

4.2 Parallel Applications

We use eight small and medium-sized parallel functional applications.

Choice of Applications Most of the applications we use have been adopted from the parallel part of the established *nofib* benchmarking suite [192] and from a recent study of Evaluation Strategies [167]. Applications using simple yet powerful patterns are deemed representative of a large class of task and data parallel applications [72]. We group the applications by the exploited parallelism pattern and investigate how program characteristics change across different run-time systems and architectures with varying number of PEs.

4.2.1 Divide and Conquer

Five of the applications use the *divide and conquer* (D&C) pattern, where a problem is recursively *split* into sub-problems that are *solved* and the results *combined* to form the final result. A generic D&C skeleton [236] could take a function that checks whether the problem is divisible, a splitting function, a merging function, and a function to solve an indivisible problem. If the problem is divisible, it is split into sub-problems that are solved recursively and the results are combined to produce the final result. In order to control the granularity of the computation, a *threshold* value can be used to restrict the depth of a *compute* tree resulting from parallel evaluation to a certain level from which on the problem is solved sequentially.

However, in our implementation we do not use the generic interface, but stick with the existing implementations that use similar but slightly different formulations, which mostly boil down to calling `par` to spark sub-computations.

- The *parfib* program computes the number of function calls for the recursive computation of the Nth Fibonacci number using arbitrary-length integers. This benchmark is deliberately aggressive in generating parallelism and aims at assessing thread subsumption capabilities of the RTS. This program is representative of regular D&C applications with a deep compute tree and with a single source of parallelism without nesting: both the splitting and the combining phases require two arithmetic operations on integers of arbitrary length and the sequential work is exponential. Therefore parallelism is very fine-grained, although the use of the GNU GMP arbitrary precision integer library, which implements Haskell's `Integer`s, means that basic operations are not single assembler instructions. We use $N = 50$ and a threshold of 23.

- The *worpitzky* application checks the Worpitzky identity[1], a combinatorial identity over integers, for two given arbitrary-length integers and is representative of the domain of symbolic computations.

$$x^n = \sum_{k=0}^{n} \left\langle {n \atop k} \right\rangle \binom{x+k}{n} \tag{4.1}$$

  At the top level this requires one exponentiation, one equality comparison, and a fold summing over a list of n intermediate results, which are computed in part in parallel and for the other part require two arithmetic operations and binomial computation using three factorial and three arithmetic operations. Parallel computations include a single source of parallelism and 3 arithmetic operations for both the combine and the split phase. We take 19 to the exponent of 27 and use a threshold of 10 (in terms of the second input value) as input parameters.

- The *queens* program determines the number of solutions for placements of N queens on a square NxN board so that no two queens are attacking each other. The positions are represented by a list of integers and generated by discarding unsafe positions. We use $N = 16$.

[1] `http://mathworld.wolfram.com/WorpitzkysIdentity.html`

- The *coins* program computes possible ways to pay out a specified amount from a given set of coins. The program is similar to `parfib` in the sense that the split and the combine phases require one arithmetic operation each, whilst a sequential solution requires finding suitable permutations of coins. In our case the value is 5777. The individual coins can take the following values: 250, 100, 25, 10, 5, or 1.

- The *minimax* application [163] calculates winning positions for a noughts-vs-crosses game on a NxN board up to a specified depth using alpha-beta search and *exploits laziness* to prune unpromising sub-trees and `parList` strategy to introduce parallelism. The board is represented by a list of rows of cells containing either `Empty`, `X` or `O`. We use $N = 4$ and a lookahead depth of 8.

4.2.2 Data Parallelism

Three of the applications are *data parallel*, i.e. the parallelism is exploited by simultaneously applying a function to the elements of a data structure. Explicit *chunking* can be used by the programmer for advisory granularity tuning at application level.

- The *sumeuler* program computes the sum over Euler Totient[2] numbers in a given integer interval, uses *chunking* for granularity control and is fairly irregular, as depending on the chosen interval granularity can vary by over an order of magnitude and is non-monotonic[3]. All the parallelism is generated in the beginning of the execution. We use interval from 0 to 100000 with a chunk size of 500.

- The *mandelbrot* application computes the Mandelbrot fractal set for a given range and image size as well as number of iterations. The application is fairly irregular due to the difference in the amount of required computation in different regions of the image. We use the image region between -2.0 and 2.0, 4096×4096 pixels image size, and 3046 iterations;

[2] `http://mathworld.wolfram.com/TotientFunction.html`
[3] but in general it increases with `n` from the specified interval

- The *maze* program is a nested data-parallel search application which searches for a path in a maze. Speculation is used to prune some of the unpromising solution candidates. Once a solution is found, the program terminates, resulting in non-deterministic behaviour due to use of randomisation in the work-stealing scheduler, whilst the computed result remains deterministic. We use the maze of size 29×29, specified via a single input parameter.

4.3 Application Characterisation

This section presents the experimental design and evaluates the results to characterise the dynamic behaviour of the applications, with a particular focus on load balancing and data locality, using the following metrics:

- time elapsed and speedup reflecting performance and scalability,
- sparks and thread counts showing available and actual degree of parallelism,
- thread sizes to assess application granularity distribution,
- heap residency as a proxy for application's working set size,
- allocation rate as a measure for heap activity,
- along with the percentage of garbage collection,
- size of the global indirection table reflecting inter-PE sharing,
- number of communication messages transmitted as a measure of communication overhead.

Elapsed time provides a direct measure of application performance, whilst speedup allows to assess scalability with increasing number of PEs. In contrast to these ends-based metrics, the means-based metrics help to explain the changes in behaviour and provide information about different RTS components. The available degree of parallelism places an upper bound on parallelism that could be exploited, whilst

the actual degree of parallelism refers to the number of converted threads. Granularity information provides insight into sizes as well as fetch and block counts and times for each thread. Heap residency, allocation rate and percentage of GC provide information on the memory management behaviour, whist global indirection table size indicates the usage of the virtual shared memory overlay (i.e. the amount of inter-PE sharing). Additionally, the communication behaviour can be assessed by using the numbers of transmitted messages grouped by message type.

4.3.1 Experimental Design

We report application performance and profiles from a median run out of three on a multi-core and on a cluster of multi-cores. We report relative speedup as we are primarily interested in the behaviour of the parallel applications.

The 48-core machine (`cantor`) consists of four AMD Opteron processors with two NUMA nodes with six 2.8GHz cores each. Every two cores share 2MB L2 cache and all six cores on a NUMA-node share 6MB L3 cache and 64GB RAM, a total of 512GB. Memory access latency differs by up to a factor of 3 depending and the NUMA regions involved with average latency of 16ns (measured using `numactl -h`). Although our primary focus is on distributed memory architectures, we use a NUMA machine for a two-fold comparison, GUM on shared-memory machine versus GUM on a distributed-memory machine, as well as to compare GUM against the established GHC-SMP RTS, which is tuned for multi-cores [169]. Moreover, NUMA architectures can be viewed as distributed-memory architectures with a very fast interconnect and thus offer an interesting design space point for comparison.

The `beowulf` cluster comprises a mix of 8-core Xeon 5504 nodes with two sockets with four 2GHz cores each, using 256 KB L2 cache, and 4MB shared L3 cache and 12GB RAM, and 8-core Xeon 5450 nodes with two sockets with four 3GHz cores each, using 6MB shared L2 cache and 16GB RAM. The machines are connected via a commodity Gigabit Ethernet with average latency of 0.15 μs.

On all the machines we run CentOS 6.5 and use run-time systems based on GHC 6.12.3, gcc 4.4.7, and PVM 3.4.6. We use the somewhat dated GHC version since

we have not yet ported the GUM RTS and profiling support to a newer version. However, we did run a small set of experiments using GHC 7.6 on the 48-core machine, which showed improved scaling for SMP, but the overall trends remained unchanged.

4.3.2 Performance and Scalability

For each application we fix the input size and increase the number of PEs to assess *strong scaling*. Figure 4.1 presents the run times on up to 48 cores on `cantor` and up to 64 cores on the `beowulf` cluster. Note the different scales.

We observe that for most applications the run time decreases as the applications are able to profitably exploit some parallelism resulting in an *order of magnitude reduction in execution* time for 5 out of 8 programs. The exceptions are `queens`, due to excessive memory use as discussed in Section 4.3.4, and `maze`, which generates more work with increasing PE numbers as more sparks end up being converted into threads. Moreover, scalability is poor for most GHC-SMP[4] runs on higher numbers of PEs, which indicates a system-level scalability issue. Surprisingly, in most cases GUM outperforms SMP on a NUMA multi-core server although primarily designed for a distributed-memory architecture.

We observe strong scaling for `parfib` and `coins` for GUM with efficiency of over 70% on `beowulf` and over 50% on `cantor`, and good scaling for `sumeuler`. However, it seems to have load balancing issues for high numbers of PEs as the number of converted threads already almost reaches a limit on 32 PEs (see Table 4.1).

Programs run using SMP show the best performance for low to medium PE numbers, whilst 48-cores results in a slowdown for 5 programs due to a memory management issue discussed in Section 4.3.4. Surprisingly, `maze` doesn't scale on SMP although it creates work proportional to the number of PEs.

By contrast, programs using GUM scale up to 64 PEs in most cases, although often the benefit of adding further PEs decreases due to increasing overhead and reduced work per PE. In particular, `queens`, `mandelbrot`, and to a lesser extent

[4]we use SMP and GUM as a shorthand for GHC-SMP and GHC-GUM, respectively

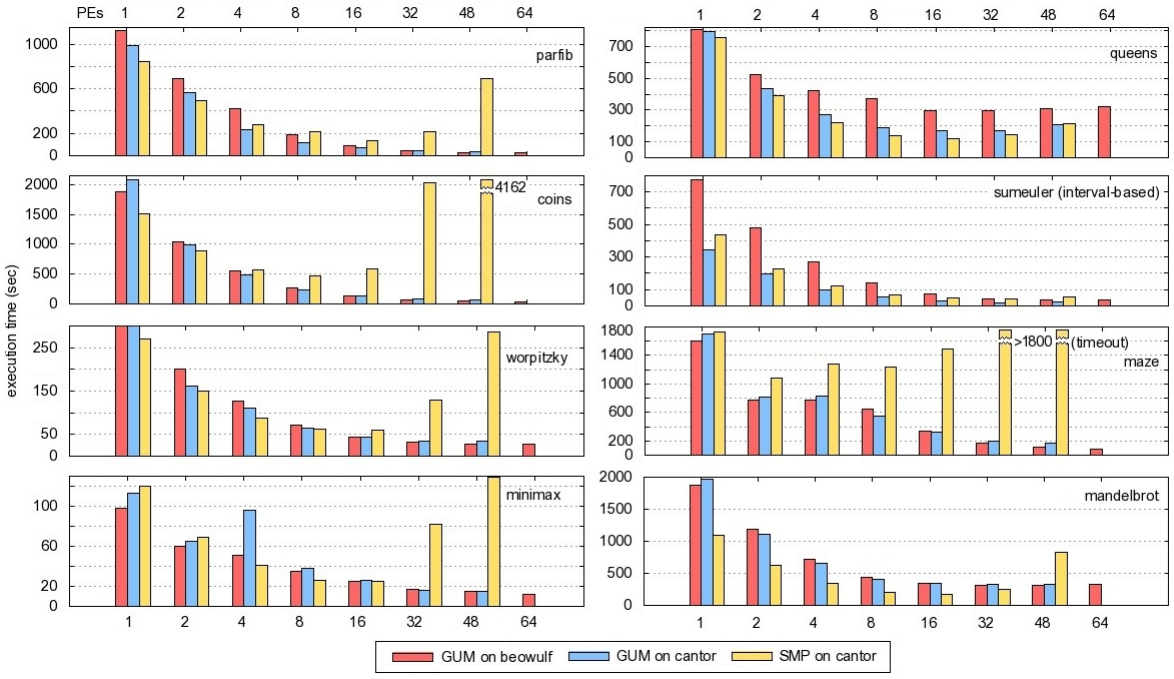


Figure 4.1: Application Execution Times (lower is better)

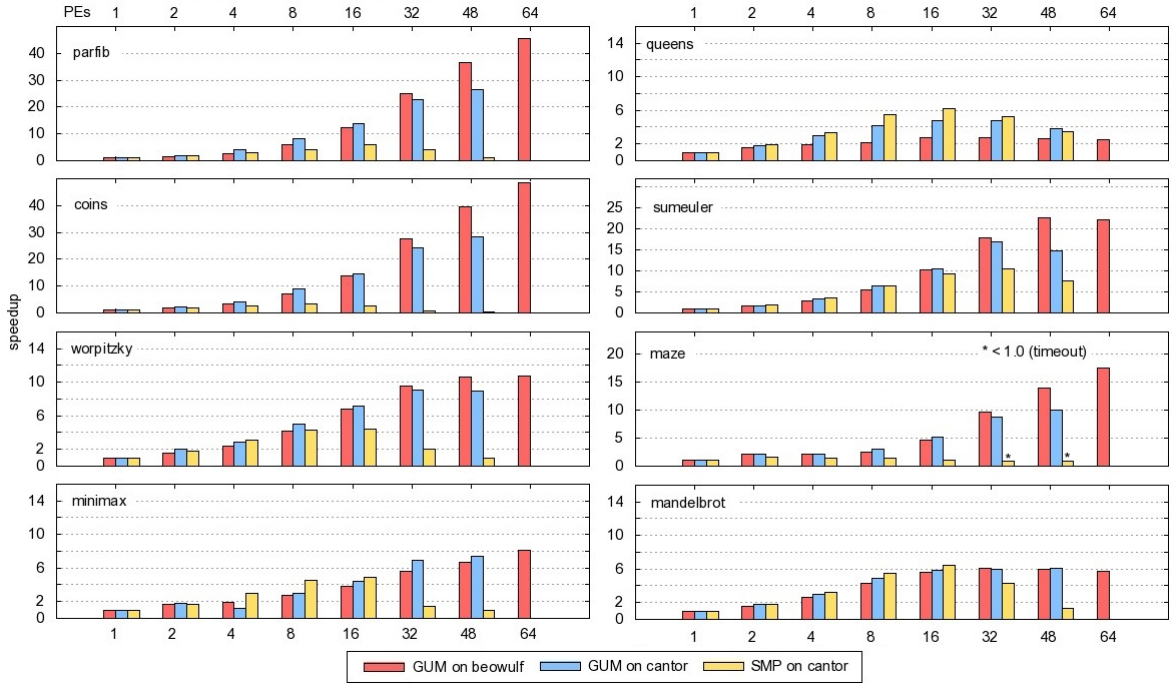


Figure 4.2: Application Scalability (higher is better)

`minimax`, exhibit limited scalability due to excessive heap residency and communication, as discussed in Sections 4.3.4 and 4.3.5, which hints at improvement potential at application level. Likewise, we believe that increasing granularity would also improve performance of `worpitzky`, since currently median thread size for this

application is very small[5] (see Section 4.3.3) and the number of threads very high (see Table 4.1).

Table 4.1: Parallelism Degree: Actual vs Potential

application c=cantor;b=bwlf G=GUM;S=SMP	number of threads on N cores 2	4	8	16	32	48	64	total sparks
sumeuler-Gb	128	165	184	192	196	198	198	200
sumeuler-Gc	135	171	186	193	197	197	-	200
sumeuler-Sc	2	4	8	16	32	48	-	200
minimax-Gb	10	30	62	143	318	449	451	1480
minimax-Gc	12	161	69	139	269	410	-	1480
minimax-Sc	5	31	92	115	127	170	-	1480
queens-Gb	10	66	135	293	544	932	1065	2462
queens-Gc	14	57	146	281	521	809	-	2462
queens-Sc	5	69	153	201	237	261	-	2462
mandelbrot-Gb	763	1259	1772	2179	2321	2333	2321	4096
mandelbrot-Gc	734	1245	1823	2261	2445	2406	-	4096
mandelbrot-Sc	1537	782	7224	22128	31999	57135	-	4096
parfib-Gb	4	34	82	328	705	1011	1733	514228
parfib-Gc	4	38	89	231	668	1834	-	514228
parfib-Sc	20	136	881	11127	54432	74347	-	514228
coins-Gb	16	62	170	591	1670	4170	5194	3507939
coins-Gc	13	36	170	633	1915	3130	-	3507939
coins-Sc	3	93	3687	7478	14784	18753	-	3507939
worpitzky-Gb	57	403	1449	4246	12510	20866	-	7340004
worpitzky-Gc	55	401	1430	4508	12728	20130	-	7340004
worpitzky-Sc	162	565	12113	49979	248115	324138	-	7340004
maze-Gb	4	13	56	412	1131	2446	4029	varies
maze-Gc	4	15	116	454	750	1682	-	varies
maze-Sc	3	52	1172	3029	timeout	timeout	-	varies

Degree of Parallelism We observe a wide range of *actual* and *potential* parallelism degrees across applications, as shown in Table 4.1. The `sumeuler` application only has 200 sparks which appears insufficient to keep all the PEs busy. This is a rare case across the benchmarks and mainly due to parameter settings. For `coins` and `worpitzky` there are four orders of magnitude more sparks available, most of which are pruned at run time. For instance for `coins` on 48 PEs, less than one percent of sparks are converted, showcasing that the system can handle large amounts

[5]using GUM the mean thread size for `worpitzky` is over an order of magnitude smaller than the second smallest mean thread size for `parfib`

of potential parallelism.

Overall GUM appears well-suited for D&C applications [159] and is able to subsume threads to a larger extent than SMP which creates threads more aggressively. In particular on `cantor` using 48 PEs, for `coins`, `worpitzky` and `parfib` SMP has 6×, 16×, and 41× more threads, respectively. This way, the RTS automatically adapts the degree of actual parallelism to the number of available PEs. For other applications, the factors range from 24× for `mandelbrot`, to 0.24×, 0.42×, and 0.32× for `sumeuler`, `minimax`, and `queens`, respectively. Parallelism is often over-abundant and fine-grained in functional programs, leaving considerable parallel slackness and emphasises the need for an effective thread subsumption mechanism.

In contrast to the D&C applications which represent a tree-like computation, thread subsumption is less effective in data-parallel applications, such as `sumeuler`, especially those where all the parallelism is created at the start of the computation. Hence, to exploit the subsumption mechanism, data-parallel applications seem to require some nesting in creating parallelism.

4.3.3 Granularity

We have extended run-time profiling capabilities of GUM and SMP to record thread granularity information (cf Section 3.3.2) and more details on the time threads spend fetching and blocking. The profiling overhead is negligible as it involves counters. In contrast to GHC, GUM RTS instances maintain private heaps and thus avoid GC-related synchronisation overhead.

In Figures 4.3 and 4.4 we present the granularity profiles of the applications. Logarithmic scales are used across both dimensions for comparability, because of orders of magnitude differences in thread sizes and numbers. For each application-architecture combination a histogram plots the distribution of threads differentiated by thread size. The x-axis presents the intervals of run times (in milliseconds) for threads that are counted on the y-axis. It therefore visualises how many small, medium, and large threads have been generated. The sub-graphs are grouped by row based on the architecture: GUM runs on Beowulf are presented in the first row

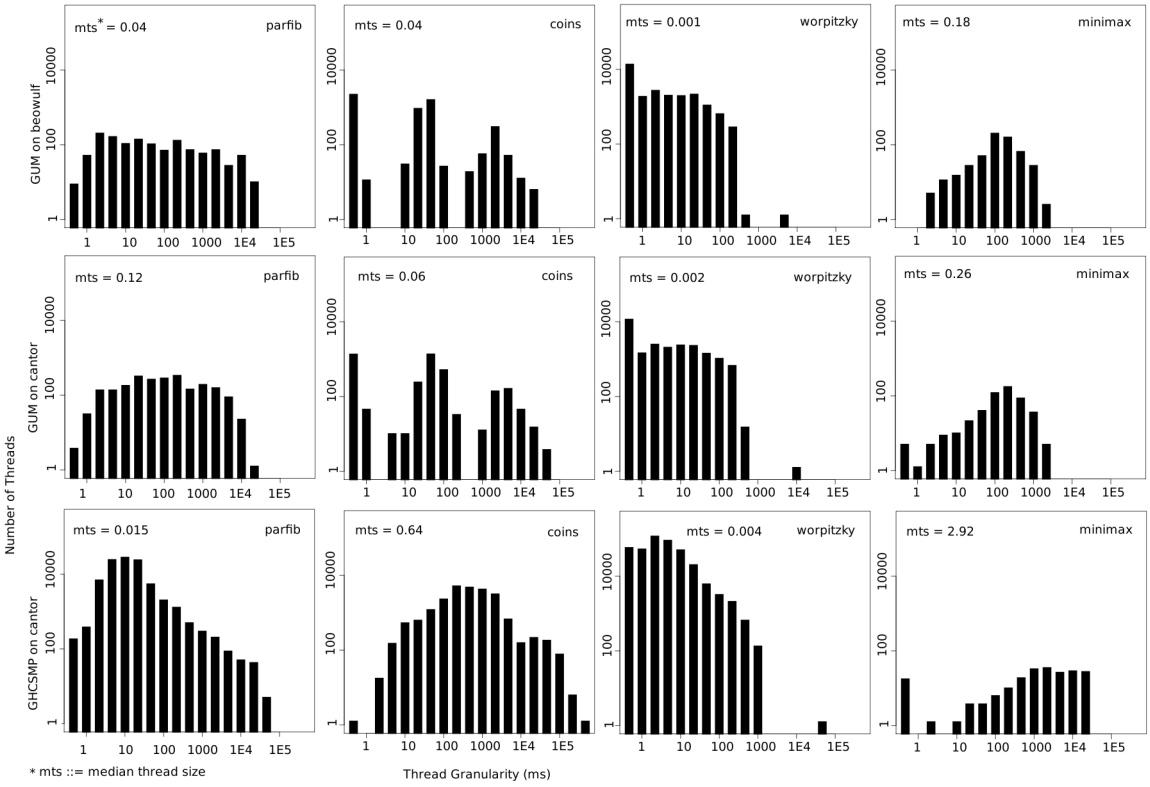


Figure 4.3: Distribution of Thread Run Times in ms (GUM vs SMP on 48 PEs)

of sub-figures, followed by GUM runs on cantor, and, lastly, by SMP runs on cantor.

For `parfib`, `coins`, and `worpitzky`, we observe an order of magnitude fewer and larger threads for GUM than for SMP, which demonstrates the effectiveness of GUM's thread subsumption mechanism and aggressiveness of SMP's thread creation for D&C applications. An interesting case is the flat, data-parallel `sumeuler` for which we see the opposite picture, as all of the work is created at the beginning of execution and almost no thread subsumption can take place. Similarly, for `minimax`, `queens`, and `maze`, SMP has fewer and larger threads. However the performance is poor for these programs, due to effects of memory use, sharing, and communication discussed in Sections 4.3.4 and 4.3.5. Moreover, `queens` and `mandelbrot` are the only applications where the execution using the SMP RTS outperforms the runs that use GUM by a small margin, due to differences in communication costs as discussed in Section 4.3.5.

In general, the shapes of the profiles for GUM on shared-memory architectures are similar to the shape of the corresponding profile on a distributed-memory architecture, but remain distinctively different for SMP profiles. This suggests that

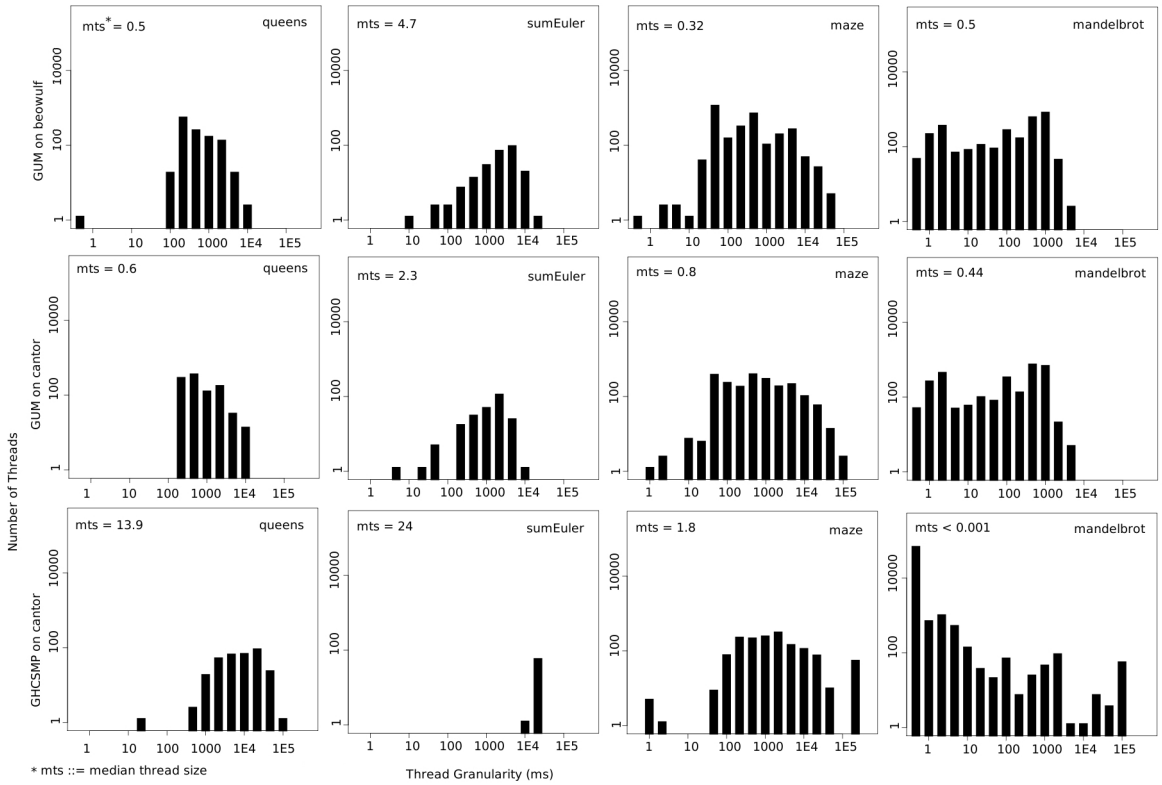


Figure 4.4: Distribution of Thread Run Times in ms (GUM vs SMP on 48 PEs) (contd.)

RTS characteristics have a strong influence on the granularity profile, especially if the architectural features are not explicitly taken into account. SMP results in more and finer-grained threads due to more aggressive thread instantiation, as seen in Table 4.1. This aggressiveness can be attributed to the SMP allowing direct access to other PEs' spark pools, whereas in GUM work stealing is used even if the PEs are located on the same physical node, indirectly acting as a throttling mechanism that allows for more subsumption to take place. For the more scalable applications we observe fewer and larger threads for GUM than for SMP. In other cases performance is relatively poor and we observe more very small threads in addition to higher memory use and communication overheads described in the following two sub-sections.

4.3.4 Memory Use and Garbage Collection

Many lazy parallel functional programs are memory-bound as they perform graph reduction that involves frequent heap operations. We measure:

- heap residency to represent a program's *working set*[6],
- allocation rate as a characteristic representing *heap activity*,
- median percentage of elapsed time used for garbage collection,
- number of global references as a proxy for inter-PE sharing.

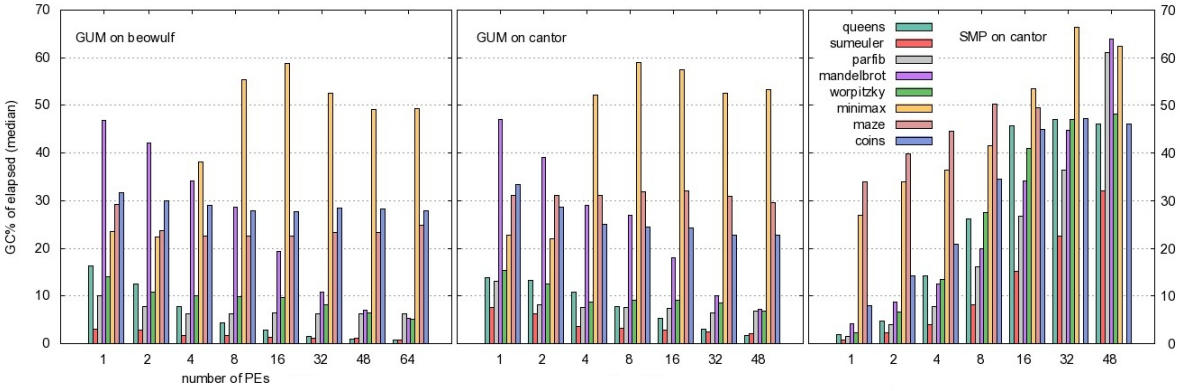


Figure 4.5: Garbage Collection Overhead

Figure 4.5 depicts the percentage GC takes and reveals a reason for scalability issues observed with SMP. The GC% increases consistently across all applications for SMP and results in severe contention on the first generation heap. By contrast, GUM initially starts off with higher GC% which then reduces or at least remains roughly constant in most cases, with the exception of `minimax` where heap residency is very high. This highlights one benefit of a distributed-memory design on shared-memory architectures by avoiding some of the synchronisation, which pays off particularly for applications with low communication rates. Moreover, in the cluster setting with N PEs $N\times$ the heap is available often reducing the need for GC on each core. GC overhead is increased for larger numbers of active threads due to resulting larger working set sizes.

In addition to GC%, the *allocation rate* signifies the *heap activity* of each application as shown in Figure 4.6. GUM maintains the allocation rate with growing PE numbers for most applications on `beowulf`, whereas on `cantor` allocation rate drops at 32 PEs indicating reduced relative heap activity due to GC overhead, as quantified in Figure 4.5. After an initial rise in the allocation rate on SMP, which

[6]which is larger when there are more active threads

confirms the benefit of more aggressive thread creation on lower number of cores on `cantor`, we observe a rapid drop for higher PE numbers, which correlates with a large increase in GC% that leads to decreased system productivity[7] and points to a scalability issue as heap activity drops significantly below the sequential level.

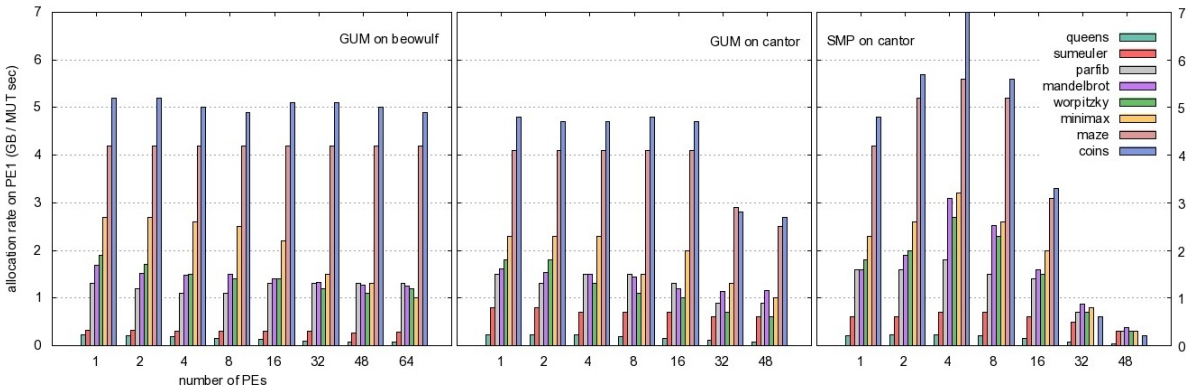


Figure 4.6: Allocation Rates

Application *working sets* are represented by *heap residency* in Figure 4.7. Note the different units across applications. We observe roughly constant or decreasing residency for GUM on both distributed-memory and shared-memory architectures, except for `minimax`, whilst for SMP the residency is growing in most cases due to higher number of live threads that refer to parts of the graph (cf Table 4.1).

Increased number of live threads leads to more roots for GC and more heap data that needs to be collected leading to increased GC frequency assuming that the avaialble heap size is constant.

This results in increased GC%, as due to contention some of the heap-allocated objects are retained for longer, an effect most pronounced for `queens` and `coins`. The jump in residency from one to two PEs for GUM is the result of sharing and the need to maintain global addresses. This reflects the potential for optimising `queens` by reducing the amount of sharing.

Moreover, as GUM uses virtual shared memory, each instance of the distributed RTS maintains a Global Address (GA) table of stable inter-processor pointers which are used as roots for garbage collection, thus increasing the live data set. However, this table only contains the indirections that reflect data dependencies determined

[7]defined as the difference of total elapsed time and GC time

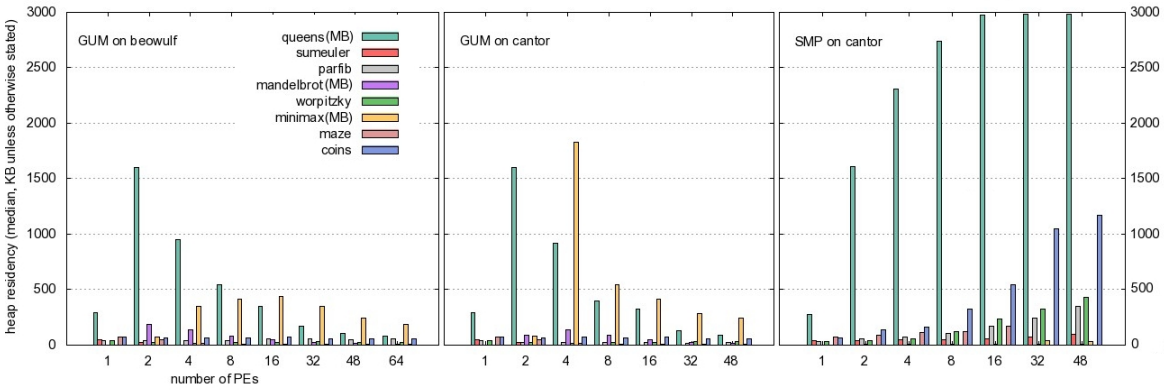


Figure 4.7: Heap Residency

by the structure of the program.

Thus, *fragmentation of the shared heap* can lead to decreased performance since excessive sharing results in higher *GA residency* and reduced locality, which leads to additional communication overhead. Thus, GIT table size, in particular GA residency, can be used as an indicator of poor locality. This hints at RTS-level optimisation potential as the location of the data structures is available inside the RTS.

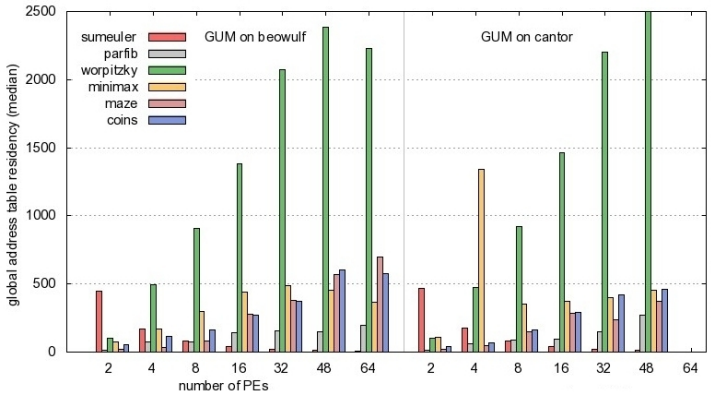


Figure 4.8: Global Address Table Residency (Heap Fragmentation)

Based on this metric, our application set can be partitioned into two classes: most of the applications shown in Figure 4.8, exhibit a moderate GA residency of at most 600 per PE. By contrast, `worpitzky` reaches a value of 2500 for a large number of PEs. Even worse `mandelbrot` (not shown in the Figure as those are an order of magnitude larger) reaches a GA residency of 8000, and `queens` (not shown) reaches a GA residency of 250000. The latter two programs exhibit a high GA residency already for low numbers of PEs, with decreasing residency when the number of PEs

increases. This points to a high degree of sharing in the program, which incurs a lot of communication and becomes a bottleneck for parallel performance.

These three programs appear to suffer most from heap fragmentation, which explains, along with very fine granularity, the limited scaling on larger number of PEs (see Figure 4.2). This underlines that the GA residency metric is a good indicator of poor data distribution in an application, which also relates to communication overhead discussed in the following Section 4.3.5. The GA residency grows for `worpitzky`, since the granularity is very fine and many sparks are stolen, which creates many GAs.

4.3.5 Communication

GUM-specific communication characteristics provide additional insight into the operational behaviour of non-strict parallel functional programs (cf Section 3.3.4). These characteristics include the communication rate, defined as the total number of messages sent per second, representing communication degree, and the percentage of steal requests (`FISH` messages) as an indicator of load imbalance or lack of work.

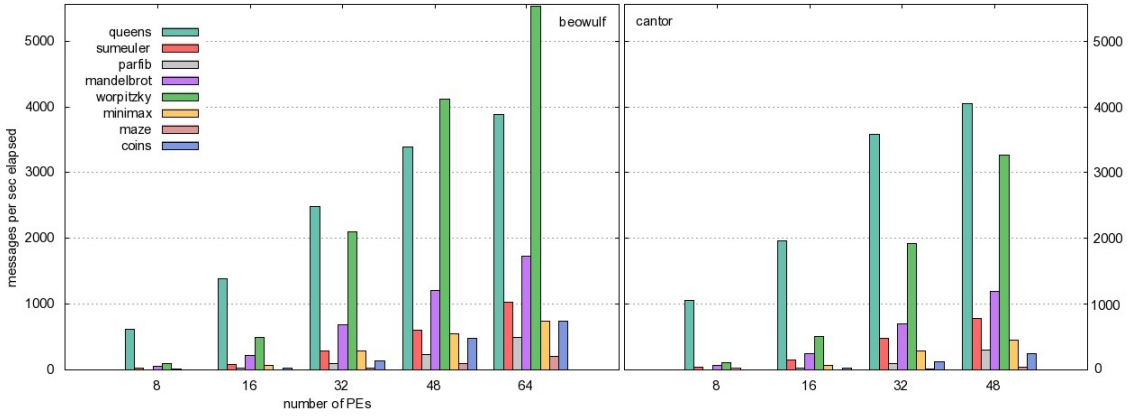


Figure 4.9: Communication Rate Comparison for GUM

As shown in Figure 4.9, for `parfib`, `coins`, `maze`, and to lesser extent `minimax` and `sumeuler`, we observe a modest linear increase in the communication rate with less than 40% of FISH messages. The median of the graph sent and GA residency increases slightly in most cases, but they are excessive only for `queens` with over 19k GA residency and around 14MB median graph sent per mutation second on 48

cores, as communication rate skyrockets to 840k messages on 48 cores with frequent very long fetches, but only 15% of messages are FISHes.

The next highest communication rate is for `worpitzky` with over 100k messages sent on 48 cores, almost 50% of which are requests for work, due to very fine thread granularity. Following suit is `sumeuler`, which exemplifies another issue — a lack of inherent parallelism for the given threshold leads to load imbalance on higher number of PEs. This is demonstrated by over 95% of sent messages being requests for work, which also coincides with decreasing memory residency and low allocation rate. For most applications the number of packets sent increases linearly and reflects the size of the shared graph, whilst the packet size is mostly very small and constant, in the range between 5 and 50 bytes, except for `queens` (4k) and `mandelbrot` (ca. 9k). We find that packets are smallest for integer-based programs and smaller for D&C programs which work on integers compared to data-parallel programs that work across data-structures. The communication rate increases with GA residency and the percentage of work requests of the total number of messages seems to indicate the degree of load imbalance.

4.4 Discussion

We have characterised a set of small and medium-sized parallel functional applications run on a server-class NUMA multi-core and on a cluster of multi-cores in terms of communication rate, heap and GA residency, allocation rate, and thread granularity. Detailed profiling of these aspects reveals diverse bottlenecks and helps gain insight into dynamic application behaviour.

First, we draw the application characterisation conclusions, where Table 4.2 provides the summary of the results for applications run using GUM ordered by scalability. We use distributed-memory results, but results on the NUMA server are similar. The characteristics are presented as categories ranging from very high to very low relative to other results rather than as absolute numbers.

Then, we discuss the conclusions from the RTS point of view, where we focus on system-level issues and discuss the observed differences between GUM and SMP.

Table 4.2: Application Characteristics using GUM on 64 PEs

application	scalability	heap residency	allocation rate	GC overhead	GA residency	communication rate
`parfib`	high	low	low	low	low	low
`coins`	high	low	high	medium	medium	high
`sumeuler`	medium	low	low	very low	very low	medium
`maze`	medium	low	high	medium	medium	low
`minimax`	low	very high	medium	high	medium	medium
`worpitzky`	low	low	medium	low	very high	very high
`queens`	low	very high	very low	very low	high	very high
`mandelbrot`	low	very high	medium	low	n/a	medium

In particular, our application characterisation conclusions are:

- Thread subsumption works well across D&C applications and architectures, as the RTS is able to handle a large number of light-weight threads and prune superfluous parallelism by merging computations into a single thread. Table 4.1 demonstrates orders of magnitude larger number of sparks than threads. Limited degree of parallelism can limit scalability as showcased by `sumeuler`, which otherwise displays favourable characteristics, but reaches only medium scalability on 64 PEs.

- From Table 4.2 we conclude that low heap residency is necessary for good performance. However, it is not sufficient as demonstrated by `worpitzky` results. Not only does performance suffer if heap residency is very high, as for `minimax`, `queens`, and `mandelbrot`, but also if communication or GA residency are high or very high.

- High GA residency indicating the degree of inter-PE sharing is strongly correlated with high degree of communication, a major overhead limiting scalability.

- Communication rate and GA residency vary considerably across applications and have a high, direct impact on parallel performance.

- Although D&C applications tend to perform better, there is no clear best parallelism pattern.

The first point empirically validates GUM's design choice with respect to thread subsumption for throttling paralleilsm degree and for granularity control, whilst the second and third support the use of private heaps and emphasise the need to avoid fragmentation of the virtual shared heap.

Moreover, we observe a medium to strong correlation[8] between GA residency and the communication rate across applications. For instance on 48 PEs *Pearson* correlation coefficient is 0.55 on `bwlf` and 0.75 on `cantor`, whilst *Spearman* correlation coefficient is 0.88 and 0.86, respectively.

From the RTS vantage point, we find:

- GUM characteristics on shared-memory machine appear very similar to the distributed-memory results, in particular with respect to granularity as shown in Figures 4.3 and 4.4.

- Compared to SMP, which allows any PE to directly access any spark pool and doesn't require communication, GUM is less aggressive in instantiating parallelism, i.e. it generates fewer threads of more coarse granularity, adapting to system latency. The higher the latency, the lazier the instantiation, as more potential threads are subsumed.

- Increased memory residency and GC-percentage in a shared-memory design limit scalability due to contention on the first generation heap, in contrast to a distributed-memory design, confirming and more percisely quantifying a similar observation from [4]. This finding is further supported by the drop of the allocation rate for SMP on high numbers of PEs as show in Figure 4.6.

- System-level information, e.g. GA residency representing inter-PE sharing and the fraction of `FISH` messages in relation to total number of messages, are potential indicators of a lack of locality and therefore are suitable parameters to control the behaviour of enhanced load distribution mechanisms.

[8]Pearson correlation coefficient reflects linear relationship, whilst Spearman relects monotonic relationship; a coefficient close to 1 or -1 is considered strong, whilst coefficient of 0 signifies that there is no relationship between the statistical variables under test

As well as flagging up a scalability bottlececk of SMP, the second-to-last point also refers to a general scalability limitation of the shared-memory designs that should reach beyond the applications studied here.

The insights from this characterisation, in particular from the third bullet point, in conjunction of with the knowledge of GUM's operational behavior (see Chapter 3) inform the design of adaptive parallelism control mechanisms, two of which are explored in the following chapters. Focusing on work distribution, we aim to extend the default work stealing mechanism to influence both the selection of victims (see Chapter 5) and the choice of sparks to be exported (see Chapter 6).

Chapter 5

History-Based Work Stealing

In this chapter, we investigate the effects of using historical information about past stealing successes and failures to improve work stealing. The key idea is to use monitored RTS-level system information to *de-randomise* work stealing. In particular, we extend the victim selection mechanism to increase predictability as well as flexibility of adaptation to system-level changes within a single application run by using monitored information that reflects the program's behaviour. The improvement aims at increasing the stealing success ratio, thus reducing the amount of communication and the associated overhead.

History-based work stealing is one of several complementary mechanisms that were identified as suitable RTS extensions based on the knowledge of the random work stealing mechanism and GUM's control model, as introduced in Chapter 3, and the results of a recent application characterisation [25], presented in Chapter 4. This chapter substantially extends the paper presented at the ICCSW'14 workshop [21].

Here we focus on the effectiveness of using information about past stealing successes and failures when selecting target PEs for work requests. The RTS is extended to maintain a list of relevant locations of past stealing successes, update it using the information carried by the enriched messages and use it as appropriate.

Additionally, we discuss the importance of selecting a suitable information invalidation interval used for ensuring that only relevant information is retained whilst stale records are removed, and the relationship between the interval, coverage and information accuracy.

5.1 Using Monitored Historical Information in Work Distribution Decisions

Using an RTS that relies on random work stealing for work distribution makes it challenging to statically predict the locations of *parallelism generators*, i.e. threads that create sparks thus increasing the degree of potential parallelism, over the execution of a given application. This is because the placement of the generators depends both on the input and the *work stealing behaviour* that is *operationally stochastic*. The behavioural variability is increased, as different runs of an application with the same input on the same target platform result in different placement of the generators across PEs, unless the programming model is prescriptive with respect to task instantiation and placement.

As a motivating example consider the non-nested data-parallel variant of `sumEuler`. We know that all parallelism will be created close to the beginning of the execution by the main PE. However, in the baseline case random stealing will lead PEs to try steal from the PEs that will not have any work and will forward on the request. We hope to improve matters by piggy-backing information on stealing successes so that the choice would be biased towards trying again where one was successful in the recent past. In our example, this would lead PEs to ask main PE again in the future after receiving a spark from it in the past, thus reducing the amount of communication.

Therefore, we extend the choice of PEs to steal from to use the available monitoring information to improve load distribution and to reduce the number of `FISH` messages, as failed work requests lead to additional messages being transmitted, which increases the communication overehad. In other words, we seek to increase the chance of choosing a PE from which stealing is more likely to succeed.

Work stealing is mainly concerned with three choices (cf Section 3.3.4):

1. victim choice for an initial `FISH` by an idle PE;
2. victim choice for a forwarded `FISH` by a false positive PE;
3. work sharing choice, i.e. which spark to export.

Except in the case of spark selection, `choosePE()` is the core function implementing the corresponding PE selection decision in the RTS and the one we extend to use historical RTS-level system information. Conveniently, points one and two above can re-use most of the code implementing the decision.

If a stealing request is not successful it is forwarded to another PE resulting in additional messages being transmitted. Our extension aims at reducing the number of sent messages and associated overhead by choosing suitable victim PEs. If the historical information is accurate and has sufficient coverage, then PEs which are ready to donate work are more likely to be chosen, thus reducing the number of forwarded requests. This can potentially decrease the total number of `FISH` messages whilst increasing the percentage of successful `FISH`es and thus of `SCHEDULE` messages, leading to the reduction of the total number of messages.

We investigate whether stealing from PEs where the most recent stealing attempts were successful yields any substantial benefits.

The key change to the baseline mechanism is in victim selection inside the `choosePE()` function: a table is maintained that records the number of recent consecutive stealing successes from a given PE or zero if the last attempt has failed. A time stamp is used to indicate how recent and reliable the information is, with zero marking the information as stale if it was not updated within the previous invalidation interval.

Logically, this can be viewed as a function that maps PEid to the time-stamped success information. In our case, it is the number of past consecutive successes or zero signifying failure.

$$getInfo(i :: PEid) \rightarrow (successInfo_i, timeStamp_i)$$

The *coverage* can be calculated as the percentage of the non-stale entries of the total number of PEs. A value close to one hundred percent indicates good coverage, whilst coverage close to zero is deemed poor. Table 5.1 shows how the stored data is interpreted to select a PE with most consecutive successes, tie-breaking on the PEid that is used as the index to access the information.

Table 5.1: Overview and Interpretation of the Stored Historical Information

information table field	value = 0	value > 0
history information	failed stealing attempt	number of consecutive successes
time stamp	information is stale	time of last update

From the implementation point of view, each PE holds two arrays of the size equal to the number of PEs and indexed by PEid. This representation is compact and favours cache locality, justifying the linear search used to find the best matching PE in the arrays. Although the array sizes grow linearly with the number of PEs, we argue that the search can in practice be viewed as a constant factor due the small maximum array size (equal to the number of PEs, e.g. 256).

The policy is expected to work best in cases where a set of parallelism generators is fairly stable over time. The overhead is low, as it involves counters, and is amortised, as the updates of the information table happen at garbage collection times (removal of stale information) and on arrival of `FISH` or `SCHEDULE` messages (update of the stored information based on the arrived information).

Additionally, the message size is increased by a small constant to carry some historical information for sharing across PEs, to increase coverage by piggy-backing in the protocol messages.

The extended mechanism falls back to random stealing if no suitable PE could be selected, either because of the lack of recent successful stealing attempts on record, or because of the information being stale, i.e. it has not changed within the last invalidation interval and is due to be purged from the information table.

Moreover, the total number of messages and the proportion of the `FISH` messages of the total message count is recorded and examined.

Balancing Accuracy and Coverage The quality of victim selection hinges on both the accuracy and coverage of the stored information. The information about the number of successful steals for a PE is deemed accurate if it is recent and truly represents the actual likelihood of successfully stealing from that PE in the system, whereas coverage is high when up-to-date information is available on most PEs.

This results in a trade-off: to ensure the largest coverage the information should be retained for longer before being invalidated; however, this may lead to stale and inaccurate or even wrong information being kept and used in making the choices. On the other hand, if the invalidation interval is set to a low value to ensure accuracy, the coverage tends towards zero, in which case the fallback baseline mechanism is likely to be used, whilst the overhead of the new mechanism is still present.

A time stamp of the last update is recorded for each PE to judge whether the stored information is reliable and to purge stale data at garbage collection times, based in the specified invalidation interval.

An additional difficulty arises from the characteristics of the application and from the dynamic nature of work stealing. Some applications exhibit flat parallelism structure with few generators, whilst other applications are more nested or recurse when generating parallelism. Thus the former generators tend to be stationary, i.e. residing on the same PE throughout the run. Whereas, in the latter case, sparks that represent the generators can be stolen resulting in different PEs becoming hosts of generator tasks at different points during the execution, which may reduce the usefulness of historical information.

Therefore, one of the main challenges is the choice of a suitable interval that leads to the highest *coverage*, i.e. the fraction of up-to-date information relative to the total number of PEs, whilst keeping the most *accurate* information. Using stale information can be misleading and reduce fishing success ratio, which may lead to severe performance degradation.

Whilst balancing accuracy and coverage appears important and would merit a separate in-depth investigation, we focus on parallel performance and communication overhead in this chapter, leaving detailed exploration of the aforementioned trade-off for future work and choose the interval using a trial-and-error heuristic.

5.2 Implementing the RTS Extension

To enable the use of historical information on past stealing successes and failures, as described above, the RTS needs to be extended in several ways. Some technical

details on extending the RTS can be found in a companion technical report [22] and in Appendix B.2.

First, a RTS option is introduced to allow the mechanism to be turned on at startup time using the `-qz<n>` option, where `n` is the *information invalidation interval* which is set to 100 milliseconds by default that is equivalent to the explicit `-qz100` setting.

Next, a per-PE data structure, in our case comprising two arrays, is added to the RTS and initialised and used to store incoming time-stamped information for each PEid as described in Table 5.1, if the RTS option is set.

Additionally, an extension of the communication sub-system is necessary to pack and unpack historical information into `FISH` messages and to forward most recent subset of the information in `SCHEDULE` messages. In our case, five extra slots for PEids of most recent successes are added.

Finally, the work stealing logic in the `choosePE()` function is extended to use the historical information for PE selection and the scheduling mechanism is extended to update the information table when new messages arrive, whilst invalidating some entries based on the value of the invalidation interval.

The previously extended profiling sub-system already provides relevant information on the number of `FISH` and `SCHEDULE` messages that allows us to assess the new policy in comparison with the baseline random work stealing.

The implementation is localised and does not require changes to the compiler. The advantage of the RTS-level implementation is that the application source code requires no changes. The only required step for the application programmer is to recompile the RTS and the applications so that they link to the new version of the RTS that supports the new mechanism.

5.3 Empirical Evaluation

We report the elapsed application run times, the absolute speedups, the numbers of `FISH` and `SCHEDULE` messages, as well as the total number of messages from respective runs with the run time close to the median run time. As we aim to reduce applica-

tion execution time by reducing communication overhead and increasing utilisation, the latter parameters are represented in terms of directly measurable numbers of messages and the number stealing successes. From these we can calculate stealing success ratio as the fraction the `SCHEDULE` messages represent in percent of `FISH` messages. Ideally, each `FISH` would lead to immediate `SCHEDULE` as a response, whilst in practice `FISH`es often travel over multiple hops before finding some work and sometimes expire and return to origin.

5.3.1 Methodology

We operationalise performance as execution time and use other observable metrics for explanation. We measure *elapsed* time in seconds, which includes both garbage collection and mutation time, i.e. time spent on evaluation of expression representing the computation leading to the final result. Times are recorded using RTS-level wrapper functions around standard OS timing functions and are reported if the related profiling flag is turned on (`-qPg` flag; see Chapter 3). This ends-based measure represents application performance and scalability, whilst means-based metrics, such as the number of `FISH` and `SCHEDULE` messages in relation to the total number of messages, are necessary to explain the results of comparing the baseline random work stealing mechanism to the extended mechanism in more detail.

We report summarised results of 32 runs for five applications run on increasing number of PEs (from 64 to 256 in steps of 16). Boxplots were chosen for visualisation as they show the summary of all the available data [256] including variability as the box size, as opposed to choosing only few runs and a mean or a single best run time for comparison, which does not include any measure of dispersion and is more likely to be affected by outliers, in particular if the number of runs is very low [130]. We settled for 32 samples per configuration, because of the stochastic nature of work stealing and the noisy environment of a shared cluster, and to improve confidence in our results, whilst often smaller number of samples are reported in the literature.

5.3.2 Target Platform

The 32-node Beowulf cluster comprises a mix of 8-core Xeon 5504 nodes with two sockets with four 2GHz cores, 256 KB L2 cache, 4MB shared L3 cache and 12GB RAM, and 8-core Xeon 5450 nodes with two sockets with four 3GHz cores, 6MB shared L2 cache and 16GB RAM. The machines are connected via Gigabit Ethernet with an average latency of 0.23 μs as measured by the standard Linux `ping` tool.

We use the CentOS 6.7 operating system, the GHC 6.12.3 Haskell compiler, the GCC 4.4.8 C compiler, and the PVM 3.4.6 communication library. The focus is on a distributed architecture as it is a more scalable but challenging architecture due to higher inter-node latency and hence higher associated communication costs.

5.3.3 Benchmark Applications

We use three applications from the set introduced in Section 4.2, where a more detailed description can be found. We select the applications that are most scalable and stable, with `parfib`, `sumEuler`, `coins`, representing flat parallelism.

To complement this set of benchmarks we add `parfibmap` and `parSEmap` benchmarks that represent nested parallelism.

- The nested `parfibmap` application uses data parallelism at the outer level and D&C parallelism at the inner level to compute the Nth Fibonacci number using arbitrary-length integers for each spark generated by the outer level. This benchmark is regular as the inner workloads are regular. However, some inefficiency may arise if sparks created at the inner level are stolen instead of the larger outer sparks. Input parameters specify the number of outer sparks and the arguments passed to the inner call (N and a threshold).

- The nested `parSEmap` application uses data parallelism at both levels. The application is irregular at the inner level due to the use of `sumEuler` as the workload associated with the outer sparks, whilst some regularity stems from use of the same input parameters with each outer spark. Therefore, potential imbalance is more pronounced, especially in the case where one PE steals small

inner sparks as opposed to the outer sparks, resulting in increased number of messages. The parameters specify the number of outer sparks and the arguments passed to the inner call, which include the upper bound for the interval to be used and the number of outer sparks. The chunk size is hard-coded as 1000.

Table 5.2: Summary of benchmark applications

application	regularity degree	nesting	parallelism pattern(s)	input parameters
`sumEuler`	irregular	flat	data parallel	300000 300
`parfib`	regular	flat	divide&conquer	53 38
`coins`	irregular	flat	d&c	7 9777
`parfibmap`	regular	nested	d. p. + d&c	256 32 28
`parSEmap`	irregular	nested	d. p. + d. p.	3000 512

For compilation guidelines and other details such as the list of run-time flags used, refer to the Appendix B.2 and to the companion technical report [22].

Below we present the measurement results that demonstrate the effects of using history-based stealing.

5.3.4 Results

The figures show the run time performance of the applications in seconds elapsed using boxplots to summarise both the central tendency and the dispersion of the data. Note the different scales.

Each box depicts fifty percent of the data that falls between the first and third quartiles, with the median represented by the horizontal line within the box. The whiskers show data outside the quartiles but within 1.5 times the inter-quartile range in either top or bottom direction from the box. The dots depict the outliers that fall outside the range of the whiskers.

We discuss the results and evaluate them using message counts as the main metric commonly used to represent overhead in distributed systems.

Performance and Scalability As summarised in Table 5.3, we observe that run times decrease by at least one order of magnitude compared to sequential run time for all applications, demonstrating substantial improvement in performance and scalability and thus the ability to benefit from using additional PEs.

However, for most applications the benefit from adding more PEs reduces with the number of PEs because of increased relative overheads and the lack of work due to fixed input size[1].

Table 5.3: Run Times (in seconds) and Absolute Speedups

application	sequential run time	baseline best speedup	history-based best speedup	history-based improvement
`parfib`	4945	**63** on 192 PEs	45 on 208 PEs	−29%
`coins`	7181	116 on 256 PEs	**130** on 256 PEs	+12%
`sumEuler`	12155	94 on 176 PEs	**131** on 224 PEs	+39%
`parfibmap`	171	1.36 on 240 PEs	**33** on 240 PEs	+2327%
`parSEmap`	466	2.36 on 256 PEs	**46** on 208 PEs	+1849%

We distinguish between the baseline random stealing and the enhanced mechanism based on historical information with an invalidation interval of 1000 milliseconds (`-qz1000` run-time flag), except for `parfib` where a shorter interval of 100 ms is used. Note that for the nested applications baseline performance is poor compared to the optimised sequential runs[2].

In some cases a longer interval can be beneficial but a detailed investigation of this accuracy/coverage trade-off is out of the scope of this work. We use the trial-and-error heuristic to select a reasonable interval.

Additionally, we observe poor performance of the nested applications in the baseline case exacerbated through stealing of the smaller inner sparks whilst larger outer sparks are available at the main PE. Due to lower amount of randomisation, in the history-based case we see a substantial improvement as older and larger outer sparks are stolen first leading to reduced amount of communication. A more detailed discussion can be found in Section 5.3.5 below.

[1] we examine *strong scaling* which requires fixing the input and increasing the number of PEs

[2] although we do not apply specific manual optimisations, the sophisticated compiler effectively exploits the latent optimisation potential

Divide & Conquer Applications Here we focus on flat (i.e. not nested) D&C parallelism and discuss the run time results of the regular `parfib` and of the more irregular `coins`.

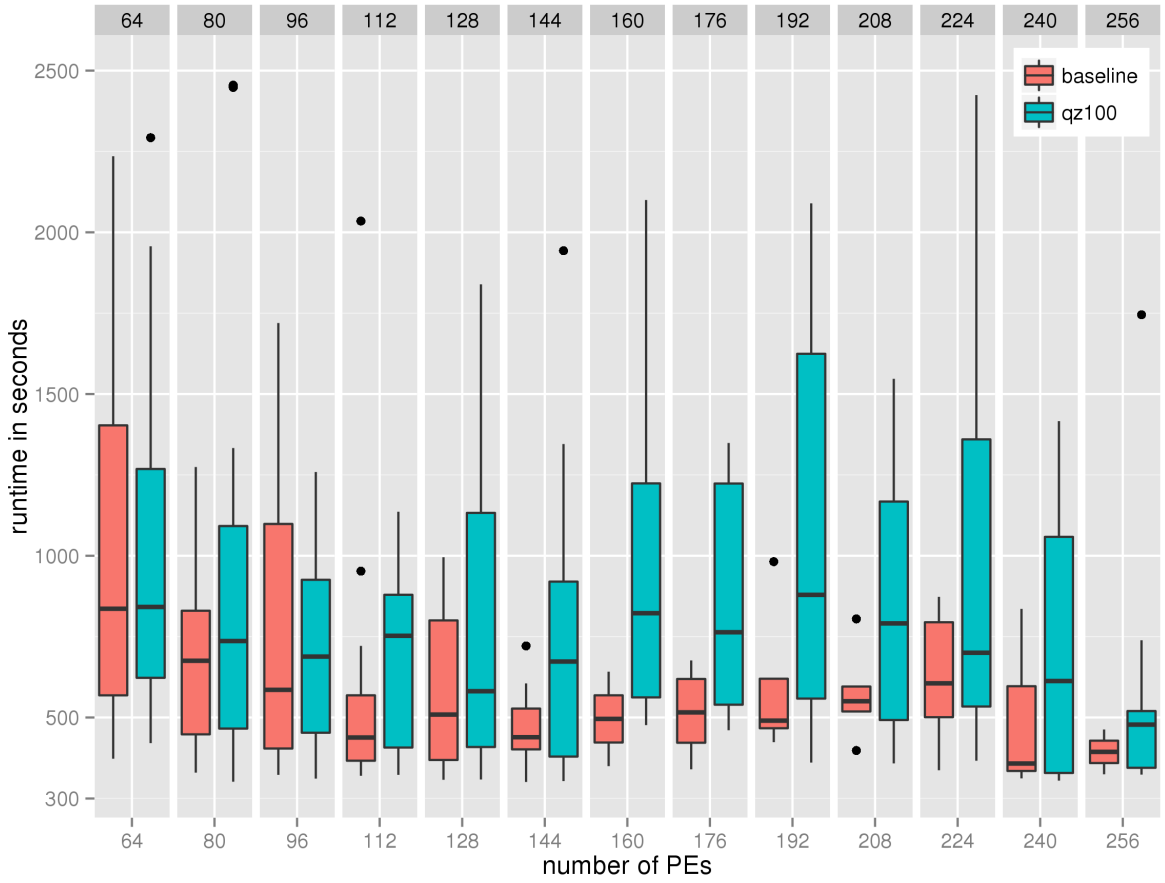


Figure 5.1: `parfib` Summary of Execution Times

Figure 5.1 summarises the data of 32 runs for each number of PEs and compares the baseline case to a case using history-based stealing. We have chosen to cut the scale at 300 seconds to reduce the amount of white space [240] and to amplify the differences for readability: 13 extreme outliers are not shown. We use a short invalidation interval as `parfib` is *flat*, in the sense that there is only one source of parallelism, and *regular*, as both sub-trees are associated with essentially equal amount of work, and thus matches perfectly the baseline execution mechanism[3].

As expected for a simple well-tuned program, we see no improvement and limited scaling with over 112 PEs in both cases, as there is little optimisation potential. The performance drop ranges from marginal to significant as the boxes overlap, as we

[3]some researchers have humorously referred to GUM as 'the parfib machine'

observe increased variability for history-based stealing for higher PE numbers, albeit visually overemphasised due to y-axis scaling. This is due to reduced stealing success as shown in Tables 5.4 and 5.5, because of dynamic parallelism generators frequently moving from PE to PE.

The results of the runs of the flat irregular `coins` application are presented in Figure 5.2. We begin the y-axis at 40 seconds to emphasise the differences and cut the axis at 320 seconds to exclude four exceptionally far-off outliers which would distort the visualisation[4].

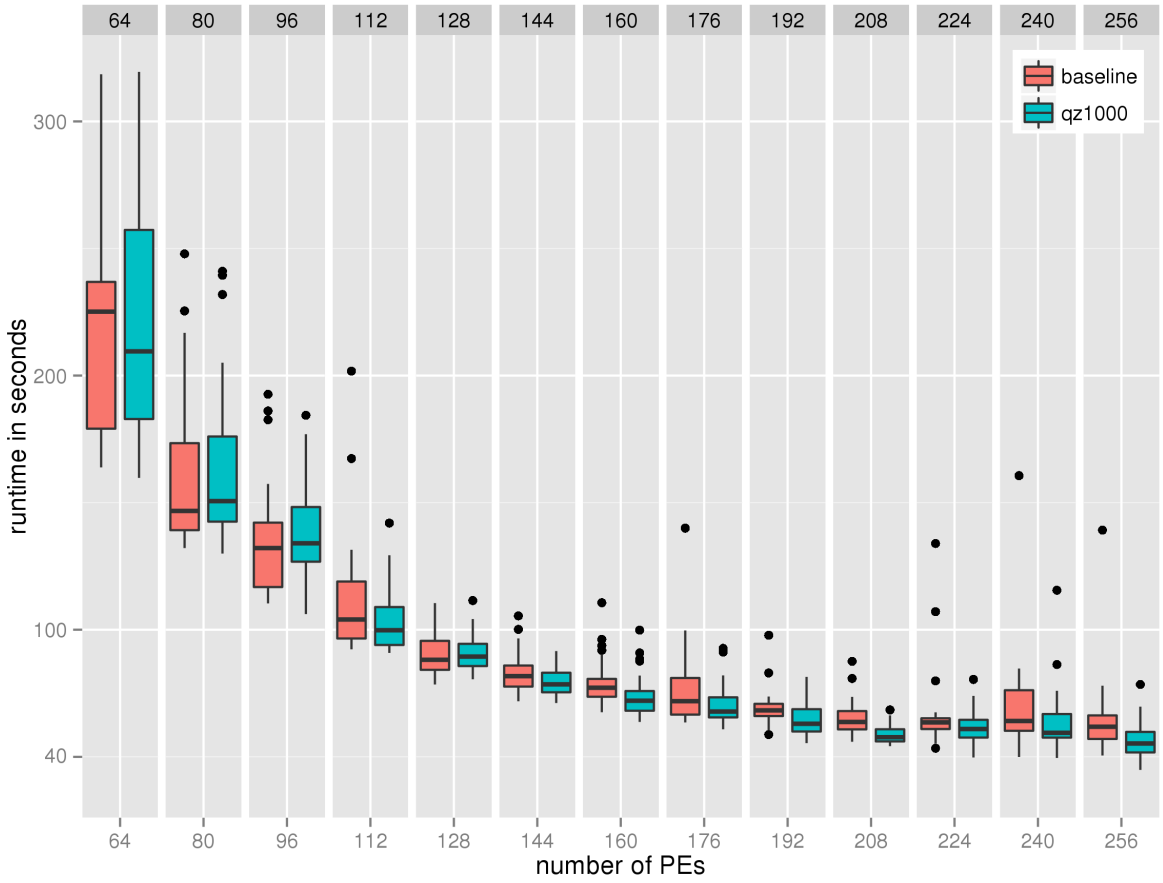


Figure 5.2: `coins` Summary of Execution Times

Despite a marginal improvement when using the history-based stealing, the difference is not statistically significant as boxes mostly overlap, except for 208 PEs case. Overall, the variability decreases in both cases as the behavior averages out with more PEs and the scaling continues to a high number of PEs (i.e. 208 for the baseline and 256 for history-based stealing).

[4]we believe the outliers are due to interference with sporadic jobs run by other cluster users

In summary, we cannot recommend the use of history-based stealing for D&C applications as new parallelism sources are created dynamically, are short-lived and tend to move from PE to PE, rendering historical information inaccurate. The new mechanism performs somewhat better for the irregular rather than the regular case, since the effects of using inaccurate information are amortised through the difference in granularity of potentially parallel tasks. The results offer some evidence for the suggestion that history-based stealing should not be used with flat regular D&C applications, whilst it appears to produce marginal improvements for flat irregular applications.

Data Parallel and Nested Applications Next we present run time results for data parallel and nested applications. Figure 5.3 shows the results for the flat irregular `sumEuler` application (note that y-axis begins at 75).

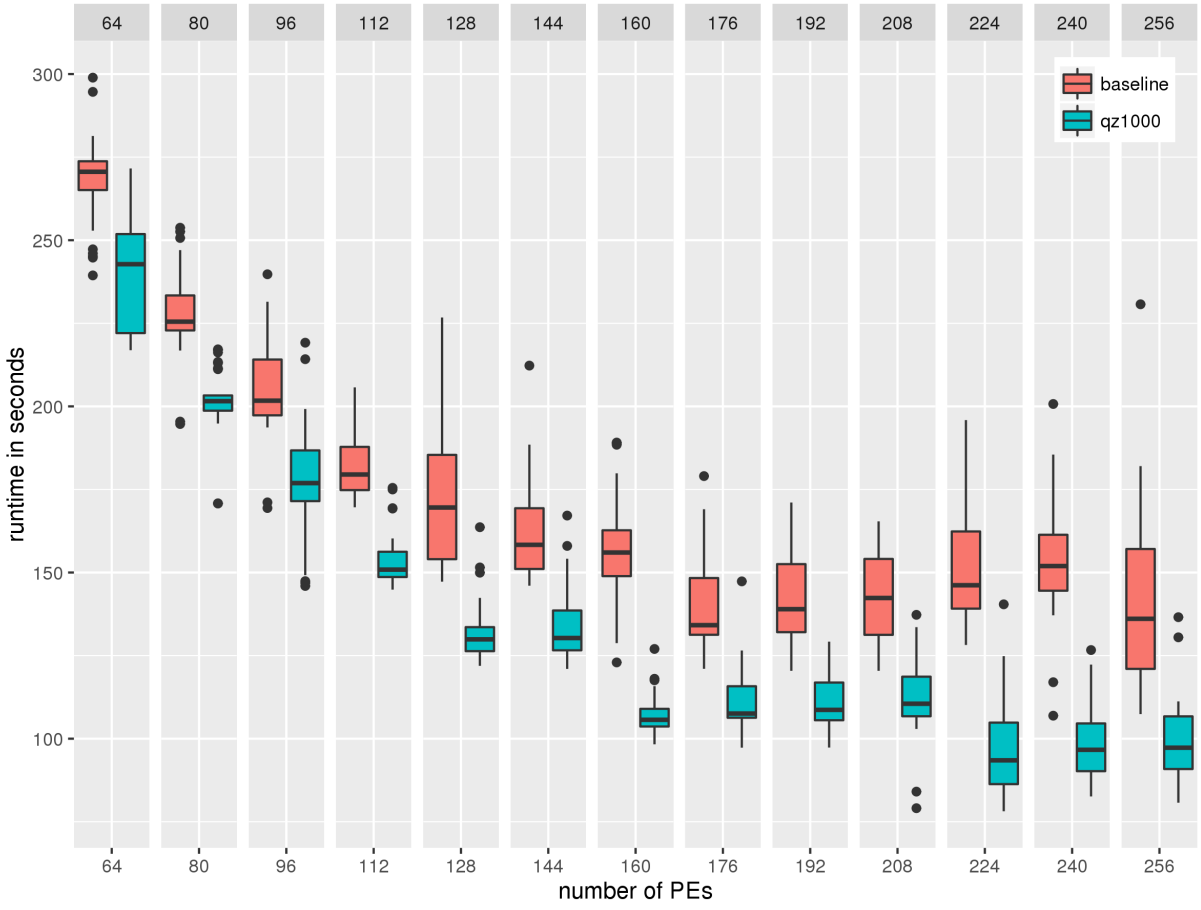


Figure 5.3: `sumEuler` Summary of Execution Times

Compared to the baseline, for history-based stealing we consistently observe

significantly lower run times, lower variability (smaller boxes and shorter whiskers) and improved scaling up to 224 PEs, as opposed to 176 PEs. We further notice that the improvements decrease with increasing numbers of PEs in both cases, which is due to the lack of work to keep all the PEs busy.

The new mechanism is effective, because in this flat application all parallelism is created by the main PE and thus past information remains accurate for a long time. This leads to a decrease in the number of sent messages whilst increasing the fraction of successful `FISH`es and therefore reducing the overhead. We evaluate the numbers of messages in Section 5.3.5 below.

Figure 5.4 depicts the run time for the nested `parfibmap` application, which uses data parallelism at the outer level and D&C parallelism at the inner level.

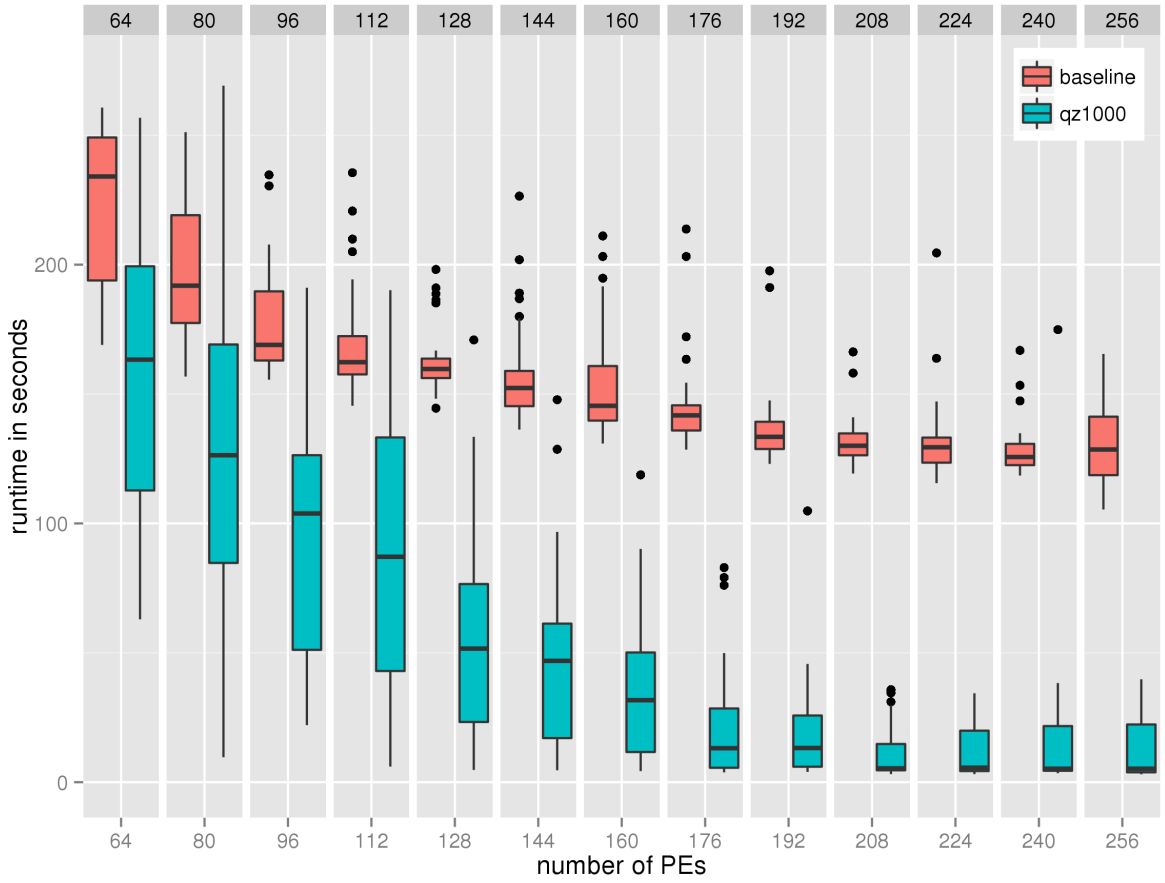


Figure 5.4: `parfibmap` Summary of Execution Times

In this case, history-based stealing outperforms the baseline case despite exhibiting higher variability, with best run times differing by over an order of magnitude. The baseline case randomly selects the victim PEs, thus neglecting the more coarse-

grained outer sparks produced by the main PE, whilst inner sparks are more suitable for thread subsumption. This is due to randomness increasing the probability of stealing an inner spark from other than main PEs, which are more likely to be randomly chosen in the baseline case.

Hence in the baseline case smaller inner sparks are more likely to be stolen than outer ones leading to an increased number of `FISH`es and synchronisation messages when parent threads depend on the results of the child sparks that were stolen. The variation is significantly lower for the baseline in most cases as fully random choices are more similar in their run-time behaviour. This is indicated by smaller box sizes in the graphs.

By contrast, in the history-based case, outer sparks are distributed first, because the information is shared about higher likelihood of successful stealing from the main PE. This leads to lower communication degree (cf Section 5.3.5). Note that the improvements are much higher than in the flat case.

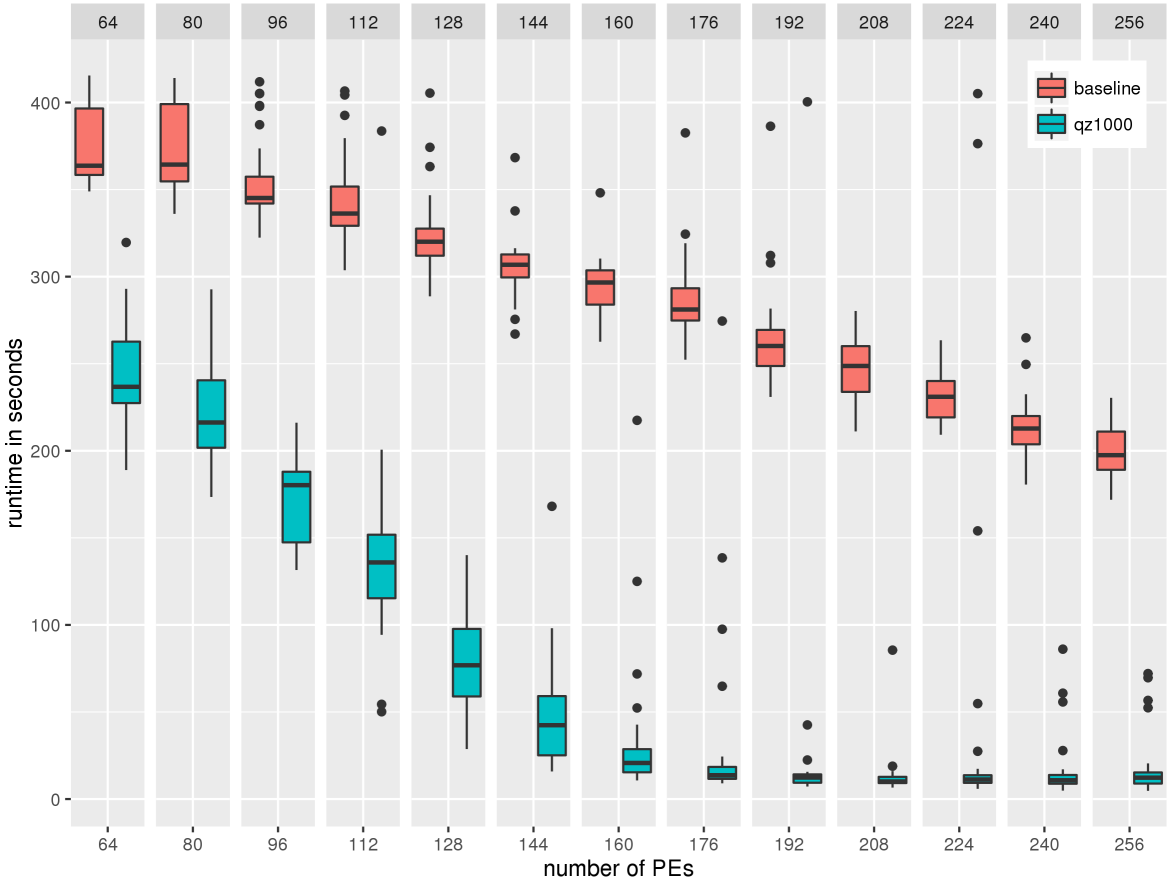


Figure 5.5: `parSEmap` Summary of Execution Times

Next we examine an instance of nested irregular applications that employ data parallelism at both levels. Figure 5.5 shows the run time performance of the nested `parSEmap` that uses `sumEuler` as the computation associated with each outer spark.

History-based stealing consistently outperforms random stealing, with over an order of magnitude difference in best run times demonstrating the effectiveness of history-based stealing. The variability is initially lower for the baseline, but from 160 PEs on it is much lower for the history-based mechanism.

Moreover, we note that from 176 PEs on, history-based stealing does not significantly benefit from additional PEs, which is due to the limited amount of work. We see that the new mechanism can substantially increase performance and scalability for nested data-parallel programs beyond the limits of the traditional approach.

In summary, on 256 PEs the run times for D&C applications do not decrease whilst the variation increases, whereas for data parallel and nested applications using historical information is consistently beneficial. In particular, the run time decreases by over an order of magnitude for the nested applications. The policy is effective for such cases, because most coarse-grained sparks are generated by the main PE and at the outer level of parallelism, hence past behaviour is predictive of future behaviour during the initial phase of the computation. By exploiting the program's behaviour the heuristic leads to reduction in communication costs and increases the work stealing success ratio, defined as the percentage of `SCHEDULE`es compared to `FISH`es, as discussed next.

5.3.5 Evaluation

Table 5.4 presents a summary of sent messages for runs on 256 PEs, the run times of which are closest to the respective median run time.

Columns two to four present the baseline data, whilst columns five to seven contain the data using the history-based stealing. For the full message profiling data refer to the tables in the Appendix A.2.

We observe significant number of transmitted messages in Table 5.4, along with high communication rates for most applications as illustrated by column two and

Table 5.4: Summary of Sent Messages (on 256 PEs)

application	baseline			history			SCHED%
	total	FISH	SCHED	total	FISH	SCHED	change
parfib	27348	17918	2352	25288	16978	2074	−11.8
coins	143303	49459	23445	201421	90844	27639	+17.9
sumEuler	750895	746901	996	429166	425163	997	+0.1
parfibmap	150161	145439	1176	21175	14278	1722	+46.4
parSEmap	97747	95325	603	21669	18691	742	+23.1

five in Table 5.5 for the baseline work stealing and for the history-based mechanism, respectively. However, this rate alone is not very meaningful, as lower run times can lead to higher rates whilst performance is actually increased.

Moreover, these results show that applications occupy different points in the communication degree space. In particular, communication rate covers a wide range from tens of messages per second for parfib, over hundreds messages per second for parSEmap to over 5000 for sumEuler.

Table 5.5: Summary of Message Ratios (on 256 PEs)

application	baseline			history			
	msg per sec	FISH% of total	SCHED% of FISH	msg per sec	FISH% of total	SCHED% of FISH	S.o.F.% change
parfib	63	66.5	13.0	52	67.1	12.0	−1.0
coins	2559	34.5	47.4	3800	45.1	30.4	−17.0
sumEuler	5441	99.5	0.1	4379	99.1	0.2	+0.1
parfibmap	1155	96.9	0.8	3529	67.4	12.1	+11.3
parSEmap	501	97.5	0.6	1667	86.3	4.0	+3.4

Only for sumEuler do we notice a decrease in both the messaging rate and in the execution time. Note that for the nested applications we observe a strong reduction in messages numbers of up to an order of magnitude and a simultaneous increase of the desirable SCHEDULE messages, compared against the baseline. Although the improvement in the percentage of SCHEDULE messages is small, from 0.1% to 0.2% it constitutes an increase by a factor of 2×.

In particular, for parfibmap the success ratio is increased from 0.8% to 12.1% corresponding to a 15.1× improvement, whilst for parSEmap the shared of FISH

messages grows form 0.6% to 4.0% corresponding to an increase of over 6.6×, which is also reflected by the improved performance. On the other hand, `parfib` data show reduction in the number of messages by 7.5% but also reduction in `SCHEDULE` messages by 11.8% as well as reduction in the percentage of `SCHEDULE`s of the total number of `FISH`es by 1% (factor of −1.08×). Similarly, for `coins` a descrease in the percentage of `SCHEDULE`s of `FISH`es by 17% (factor of −1.56×, even though the number of `SCHEDULE` messages is increased by 17.9%, which is still at a lower rate than the total increase in messages of 40.8%. This suggests that History-Based Stealing is not effective for D&C applications.

Furthermore, the `FISH` messages represent the largest fraction of the total number of messages, because for high numbers of PEs the fixed work amount leads to a lower potential work amount per PE. In particular, for the applications that benefit from using historical information, all of them have over 97% of all messages being `FISH`es, which is substantially decreased when history is used. We can see an increase in the percentage of `FISH`es in relation to the total number of messages for D&C applications (marginal 0.4% for `parfib`, and substantial 10.6% for `coins`), whilst the percentage is decreased for data-parallel and, more substantially, for nested applications (0.4% for `sumEuler`; 29.5% for `parfibmap`; 11.2% for `parSEmap`). Coupled with the improvements in the percentage of `SCHEDULE` messages in relation to the number of `FISHes`, this explains the increase in performance, as fewer messages are sent and a higher proportion of work requests are successful, thus further reducing the number of messages.

The improvements are most striking for `parfibmap`, where the total number of messages on 256 PEs for the new mechanism is smaller than the number of `FISH`es alone on 64 PEs for the baseline case.

5.4 Discussion

We find improved run time of up to an order of magnitude on up to 256 PEs, showcasing substantial scalability. This is in part due to decreased number of messages, in particular of `FISH`es for data-parallel and nested applications. For these bench-

marks we also observe increased percentage of `SCHEDULE`s of `FISH`es, signifying that the mechanism is effective in biasing PE choice to improve the likelihood of obtaining work. The mechanism is effective because much of the parallelism is initially generated by the main PE and these sparks once turned into threads also generate multiple further sparks. This is exploited through the use of historical information. However, this heuristic fails in cases where past application behaviour is not predictive of the future behaviour as it is the case for more irregular D&C applications with large number of very fine-grained threads and parallelism generators spreading across PEs. Additionally, there is further room for improvement suggested by sub-linear speedups and by the still relatively large number of messages being transmitted.

Possible threats to validity include confirmation bias, selection bias, implementation and architectural details. Whilst we have discarded some applications from our set of benchmarks, this was due to very limited scaling and unstable sequential performance, which would result in misleading results. Moreover, we contend that the five used applications cover sufficiently different points in the application space in terms of communication degree, parallelism degree and pattern, as well as application regularity and nesting. We believe that the results are meaningful as we run real code on real hardware. Although we can't fully control the external load on the shared cluster, we checked the load before starting the runs and we use 32 runs for each input-application-PE-number combination to improve confidence and avoid impact by outliers. As the variation is reduced in most cases with increased PE numbers, we believe the overall trends are likely to remain for larger inputs.

Furthermore, the mechanism is not specific to the language and the run-time system, and could be applied to other work stealing schedulers. Additionally, the absolute speedup here is not for the optimal sequential application but for a sequential elision of the parallel code. However, we observe that in most cases the compiler is able to optimise the applications well. We also do not compare the run times to C programs as we did not have the resources to develop corresponding low-level versions using Pthreads or OpenMP and MPI, which we would expect to outperform

GpH applications at a higher development cost.

The following chapter presents a complementary extension, investigating the way to influence the spark selection for export based on additional system-level information to *co-locate* sparks from the same source of parallelism to improve locality according to ancestry dependencies within the computation, made explicit through the use of a new language primitive.

Chapter 6

Colocation of Potential Parallelism

Work stealing is a popular passive work distribution mechanism where idle PEs attempt to steal work from busy PEs. We introduced general work stealing in Section 2.4.1 and its implementation in the context of GUM in Section 3.3.4.

In the simplest version the victim PEs are selected at random. Used in the Cilk [38] RTS and in run-time systems for functional languages such as Multilisp [140], and GHC-SMP [113], this mechanism often employs a FIFO policy for storing and selecting potentially parallel tasks for donation. The rationale for this choice is to favour exporting older potentially parallel work units or *sparks*, which are deemed more likely to have larger granularity and may generate further parallelism, especially if Divide-&-Conquer (D&C) pattern is used [174].

Choosing larger computations aims to offset the communication costs, in addition to latency hiding, by reducing the number of transmitted messages, in particular in computations that use the D&C parallelism or are nested, and are run on distributed architectures with very high communication costs. This is due to the resulting computation structure, where parent threads create sparks and require the results of children threads, created from these sparks, to proceed.

In this chapter we investigate the effect of *Spark Colocation* (SC), our alternative approach to choosing sparks to be donated, on performance and scalability of five applications and explain the results based on means-based metrics from execution profiles including per-PE thread activity, thread granularity information, message counts as well as the degree of sharing across PEs.

The main idea of Spark Colocation is to *improve load balancing* by donating the spark that is *most closely related to the computation performed by the thief* according to a specific similarity metric in response to an incoming `FISH` message. We use an encoding of the sparks' places in the computation tree to colocate closely related sparks, based on the computed maximum prefix matching that represents the distance between sparks.

6.1 Design

Spark Colocation extends the baseline random work stealing mechanism and aims at investigating the effect of favouring colocation of *related* sparks, rather than selecting a spark to export based only on its implicit age. The aim of this choice is to improve *data locality* and *load balancing*, which in turn would improve application performance and scalability.

Consider the example from Figure 6.1 that illustrates a situation where two PEs work on several tasks. The tree structure represents computational dependencies, whilst the dashed regions depict which tasks are located on which PE.

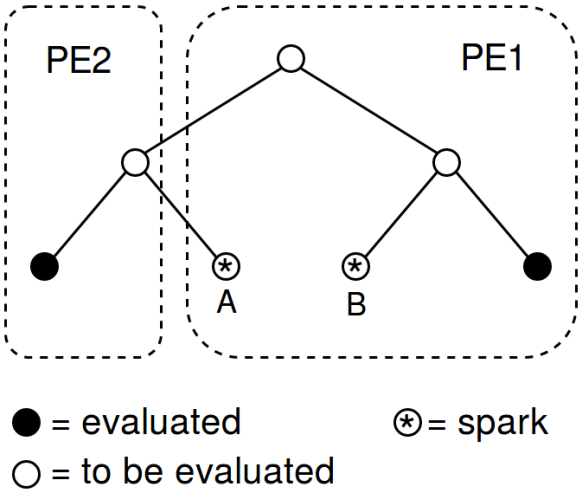


Figure 6.1: Example of Potential for Colocation

In particular, both sparks ended up on PE1. As PE2 continues the evaluation it runs out of tasks and sends a `FISH` to PE1. In turn, PE1 can now decide which spark

to donate. It would donate B, which we assume is older[1], in the baseline case. Then it would continue to execute the remaining spark A locally. However, the result of A is needed by PE2, which would require additional communication. Similarly, if spark B is exported and turned into a thread on PE2, communication is required to send the result to PE1. If Spark Colocation is used A would be donated as it is more related to the computation on PE2.

The main idea is to allocate computations to PEs that have worked on related computations. A related computation is located closely in the same computational sub-tree, because its result or produced data are likely to be required by the other computation. The concept of SC builds on the notion of proximity between computations. Two sparks are defined to be in close proximity if the path in the tree between their nodes is short. In particular, if the root node is on the path, the sparks can be considered unrelated.

This needs to be made explicit in the implementation by encoding these paths (see Figure 6.2), thus recording the information about the relationships among sub-computations, which is initially implicitly available in the source code, but is lost during compilation in the default case. Hence with SC, the encodings marking the sources of parallelism are forwarded to the RTS, which reconstructs the relationships between sub-computations and uses them to influence scheduling and load balancing decisions at run time. In our case these encodings are used to dynamically select suitable sparks to export.

Keeping all the PEs busy tends to increase the proportion of useful communication messages. The load can be represented by the number of the runnable threads at the time of measurement, whilst the size of the global indirection table (GIT) expressed through the number of Global Addresses (GAs) represents the amount of inter-PE sharing. Additionally, exporting different tasks is likely to affect the number of created sparks and the conversion rate into threads. The local reduction mechanism that includes thread subsumption remains unchanged.

Informally, the colocation algorithm behaves as follows: if a PE is idle, it will attempt to steal work from others that will respond with the spark on the path

[1]this is reasonable as PE1 is the main PE and PE2 starts with no work

through the compute tree that is most closely related to the stealing thread, rather than with the oldest.

We use this *ancestry* relation with the *maximum prefix* function as the matching function for finding the best match between the encoding of the thief and of the sparks available to the victim. If no match is found, the baseline mechanism is used by either exporting the oldest spark or forwarding the `FISH` to another PE.

One computation X is deemed an immediate parent of another computation Y if it has created the spark from which Y resulted when this spark was converted into a thread. By extension, Z is Y's ancestor if it is X's parent or ancestor. In our case, the ancestry relation is encoded as a path in the tree represented by a string of symbols that encode the branch at each tree level. Thus, the degree of relatedness between two computations can be defined by the extent their encoding prefixes match.

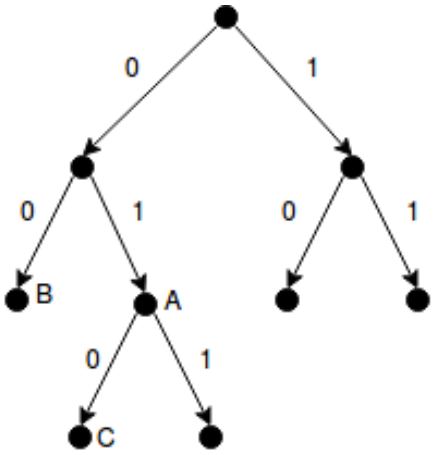


Figure 6.2: Spark Ancestry Encoding Example

Figure 6.2 illustrates the encoding for two sources of parallelism, where each choice point becomes a label in the encoding sequence. Thus according to the number of possible labels base 2 is used for the encoding in this case. Note that the nodes in the tree represent spark points and the tree is not the full compute tree.

For example, if spark A with the encoding 01 was turned into a thread and then had the choice between sparks B and C, the latter would be chosen as given its encoding 010 it has longer common prefix of length two with A as opposed to B with encoding 00, which shares only one symbol with A. We can also see that A requires

the result of computation C, whilst it does not require the result of computation B to proceed.

We select maximum prefix as a matching function because the resulting encoding mirrors closely the actual tree-like computational structure of the applications. The ancestry relation defines the distance between a thread's encoding and the encoding of a given spark. The smaller the distance the more related two sub-computations are deemed to be. An investigtation of alternative encodings and matching functions is out of the scope of this work.

6.2 Implementation

Spark Colocation is implemented in the RTS. An explicit language primitive, a version of the `par` combinator we call `parEnc`, is used *to label the sparks*. This takes additional encoding arguments that are forwarded to the RTS. The path to the spark constitutes an encoding, where we start from the root and add a symbol for each sub-branch chosen at each level. The symbol corresponds to the label attached to a `parEnc` site that lead to the creation of the spark and is appended to its inherited parent's encoding.

Note that we introduce no new concepts here: `parEnc` is just another variant of `par`. Alternatively, user-defined cost centres [110], separate from `par`, could be used. Although we use programmer-placed annotations, we argue that it is possible to automatically place such annotations by enumerating `par`s and replacing each with `parEnc`, with the corresponding encoding as an argument. All the parallelism management is performed transparently by the RTS.

Data Structures Internally, this experimental implementation employs hash tables to store and access the information on threads and sparks by mapping respective ids to information-holding data structures. This mechanism enables the RTS to distinguish sparks based on their location within the implicit compute tree of the application for a given input. A potentially more efficient implementation would store the encoding as an additional field directly in the thread descriptors (TSOs), but

this would require substantial changes to the compiler and to the garbage collector.

Finding the Best Spark Each time a spark is created it stores its full encoding in the hash table. This encoding is compared to the encoding carried by an incoming `FISH` message, extended with information about the encoding of the thread most recently executed by the thief. The spark pool is traversed and a spark with a maximum prefix match is donated. The maximum traversal length can be specified as an RTS option. In case there is no match, the oldest spark is donated. Finally, if there are no sparks available, the request is passed on to another PE. When a thread is created the data structure holding the spark's encoding information is released after the encoding has been passed on to the thread's corresponding data structure inside the hash table.

6.2.1 Spark Selection

As mentioned above, our focus is on changing spark selection, which is originally implemented as the `findSpark()` function that selects a spark for export, by traversing the spark pool.

In the baseline mechanism, the spark pool is implemented as a lock-free double-ended queue (deque) [228], so that the owning PE can add new sparks at the tail of the deque whilst sparks are exported off the head in First-In-First-Out (FIFO) manner. This mechanism avoids most of the synchronisation cost as it is only incurred when threads actually attempt to evaluate the same spark. Older sparks tend to refer to work of larger granularity and that also likely to generate more parallelism making corresponding sparks suitable for donation, whilst the youngest sparks tend to be related to the current computation and could be beneficially inlined. This is similar to the Breadth-first-Until-Saturation-then-Depth-first mechanism [49] that we discussed in Chapter 2.4.

6.2.2 Matching Function

We encode ancestry as a string of symbols to the base of the total number of `pars` in the program.

Rather than using a static call-tree which may be available at compile time, the labels attached to the spark sites are passed to the RTS which dynamically collates the ancestry information into encodings, so that it incorporates the dynamic relationship that arises at run time. This is beneficial as in a non-strict setting some parts of the graph may remain completely unevaluated if they are not demanded based on the application-input combination.

As a fitting choice, maximum prefix string matching is used to determine the spark for donation, since it represents the closest relation between the computations in the tree based on the corresponding encodings.

6.2.3 Packet Format

To propagate ancestry information between the PEs, the packet format is extended for the `FISH` and the `SCHEDULE` protocol messages. `FISH` is extended to carry the requesting PE's encoding, whilst `SCHEDULE` includes the exported spark and its encoding. Once turned into a light-weight thread, the spark's encoding is used to update the current thread's encoding, which is in turn inherited by the sparks generated by this thread.

6.2.4 Profiling

To assess the behavioural difference compared to the baseline mechanism, the event-based profiling sub-system is extended to record thread granularities in addition to the already available profiling information such as per-PE load over time, message counts, and GA residency. If event-based profiling is turned on, we record for each thread its life time from creation to destruction along with PE id in the corresponding log file. More details on this extension can be found in Appendix B.4.

The extension does not impede scalability as it only involves keeping an additional counter adding little to the existing profiling overhead, whilst the events are

written out to file as they occur using a separate asynchronous thread that is responsible for buffered I/O. At the implementation level, a small number of localised changes to the RTS is required.

6.3 Performance Evaluation of Spark Colocation

To evaluate the effectiveness of Spark Colocation we compare it to the baseline mechanism by running five applications on a 32-node Beowulf-class cluster of 8-core nodes using up to 256 cores. SC is particularly relevant for high-latency clusters, because it is designed to reduce communication costs. We describe the hardware setup in Section 6.3.2 and the applications used in Section 6.3.3.

6.3.1 Methodology

We run each application five times for each PE-count both with and without event-based profiling and compare the median runs with and without Spark Colocation. We report run times and speedups from the runs without event-based profiling avoiding the profiling overhead. The profiling runs are used to generate the graphs in Figures 6.11-6.18 (per-PE load) and 6.22-6.25 (granularity). The number of runs is relatively low due to the long sequential run time for the chosen inputs and the amount of measurement points due to variation in PE counts, which allows us to assess scalability on higher PE numbers. We use a lightly loaded cluster. However, as the cluster is not dedicated and does not use a queueing system, we can not fully rule out some variation due to interference with other processes running on the machines. As PVM is used as a communication library, processes are placed onto nodes in a round robin fashion as specified in a hostfile.

We measure the elapsed time and calculate the relative speedups based on the sequential elisions[2] of the programs to assess scalability on up to 256 PEs.

Using ends-based metrics such as elapsed (wall-clock) time and speedup alone doesn't provide sufficient insight into why the observed effects of SC take place, for instance with respect to load balance over time. Therefore, we also collect profiling

[2]the compiler obtains an elision by disregarding the `par` annotations

data for several means-based metrics: per-PE numbers of threads over time as a measure of load balance and degree of parallelism; thread sizes reflecting granularity; numbers of transmitted messages of different types; as well as sizes of internal data structures holding inter-PE pointers to assess data locality.

6.3.2 Target Platform

The applications are run on a 32-node Beowulf cluster of multi-cores using up to 256 PEs. The cluster comprises a mix of 8-core Xeon 5504 nodes with two sockets with four 2GHz cores, 256 KB L2 cache, 4MB shared L3 cache and 12GB RAM, and 8-core Xeon 5450 nodes with two sockets with four 3GHz cores, 6MB shared L2 cache and 16GB RAM. The machines are connected via Gigabit Ethernet with an average latency of 0.23 μs measured using the Linux `ping` utility (average round-trip time of 100 packets of standard size).

We use the CentOS 6.7 operating system, the GHC 6.12.3 Haskell compiler, the GCC 4.4.8 C compiler, and the PVM 3.4.6 communication library. The applications are compiled with optimisations turned on (`-O2`).

6.3.3 Benchmark Applications

We use applications from the set introduced in Section 4.2, that could be expressed using the `parEnc` notation, which results in D&C parallelism.

In particular, we use `parfib`, `parpair` with calls to `sumeuler` and `parfib` nested within the pair and evaluated in parallel, interval-based `sumeuler` reformulated using the D&C pattern, `worpitzky` and `minimax`.

Table 6.1: Applications Overview

application	parallelism pattern	regularity	input parameters
`parfib`	D&C	regular	50 35
`parpair`	nested D&C	irregular/regular	100000 10 50 35
`sumeuler`	D&C	irregular	100000 10
`worpitzky`	D&C	irregular	27 30 18
`minimax`	D&C	irregular	4 8 2

We ported the flat data-parallel `sumeuler` to the interval-based D&C version to add a second source of parallelism. Otherwise, in a flat data-parallel version with a single source of parallelism, the SC version would result in exactly the same choices at the baseline.

6.3.4 Results

We present graphs visualising the performance and scalability for the applications using run time and speedup. The run time is the end-to-end elapsed time of the application run, including garbage collection and mutation time.

Table 6.2 shows the overview of the results of using Spark Colocation on 256 PEs: substantial speedups can be reached for both the baseline as well as for the colocation case, achieving speedup improvement of up to 46% with SC.

Table 6.2: Application Speedups on 256 PEs

application	baseline speedup	colocation speedup	**change in %**
`parfib`	204	219	+7
`parpair`	200	231	+16
`sumeuler`	142	207	+46
`worpitzky`	175	101	−42
`minimax`	95	79	−17

However, we also observe a drop in speedup for SC of 17% and 42%, for the less scalable `minimax`, and for `worpitzky` with excessively fine-grained parallelism and parallelism degree, respectively.

Run Time Performance Figure 6.3 depicts orders of magnitude reduction in run time as the number of PEs increases. Note the logarithmic scale using the natural logarithm, employed due to the high differences in run times among the applications; the data for SC runs is denoted by the suffix *_sc* in the legend. The results, selected from the median run based on the performance for 256 PEs, indicate that functional programs can scale, exploiting large amounts of parallelism and additional PEs.

We consider some extreme values outliers and thus report the median of the data after outlier removal. Figure 6.4 presents an alternative view showing the

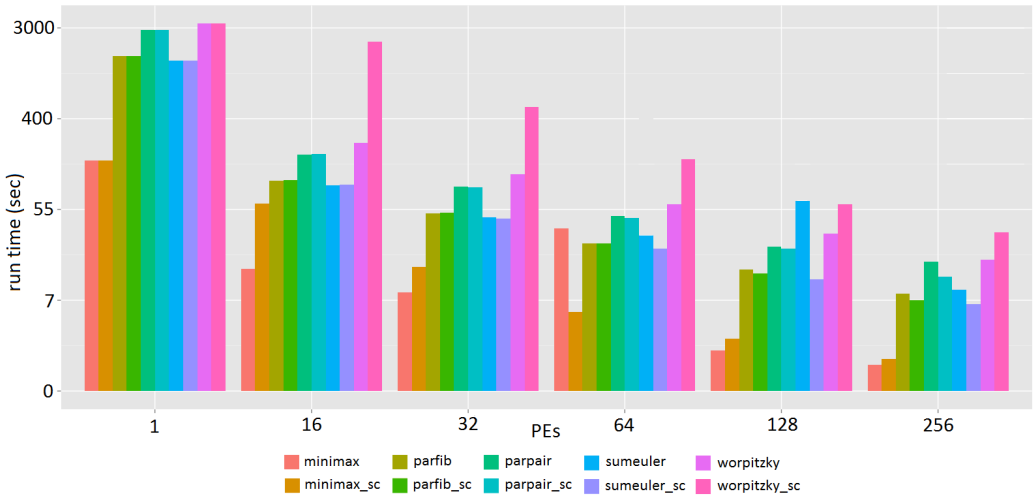


Figure 6.3: Spark Colocation: Runtimes (log scale)

percentages of change in speedup for SC using the median for each of the selected PE numbers. This way we see that the spike for `sumeuler` is an outlier. Positive values show the advantage from using SC, in particular for higher PE numbers, whilst negative change in speedup depicts the cases where baseline outperforms SC.

Summarising Figure 6.3, in particular for 256 PEs, SC leads to better performance for `parfib`, `parpair` and `sumeuler`. However, for the more fine-grained `worpitzky` and the less scalable `minimax` the baseline mechanism outperforms SC.

Scalability The scalability is assessed using *strong scaling* as the input is fixed and is not increased with increasing numbers of PEs. We report application speedups based on sequential elisions of the parallel programs that a compiler optimises automatically, given the correct flags, whilst disregarding the parallel annotations on one PE. We focus on the comparison between Spark Colocation and the baseline for different numbers of PEs with more than two PEs, because multiple PEs are required for work stealing to operate.

Figure 6.5 presents an overview of speedup, whereas Figures 6.6 – 6.10 present the scalability of each application in the baseline and the colocation case compared to ideal speedup, represented by a straight dashed line. Figure 6.5 demonstrates that all applications are able to scale achieving at least 75× speedup, and exceeding 200× on 256 PEs for `parfib`, `parpair` and `sumeuler` (with Spark Colocation).

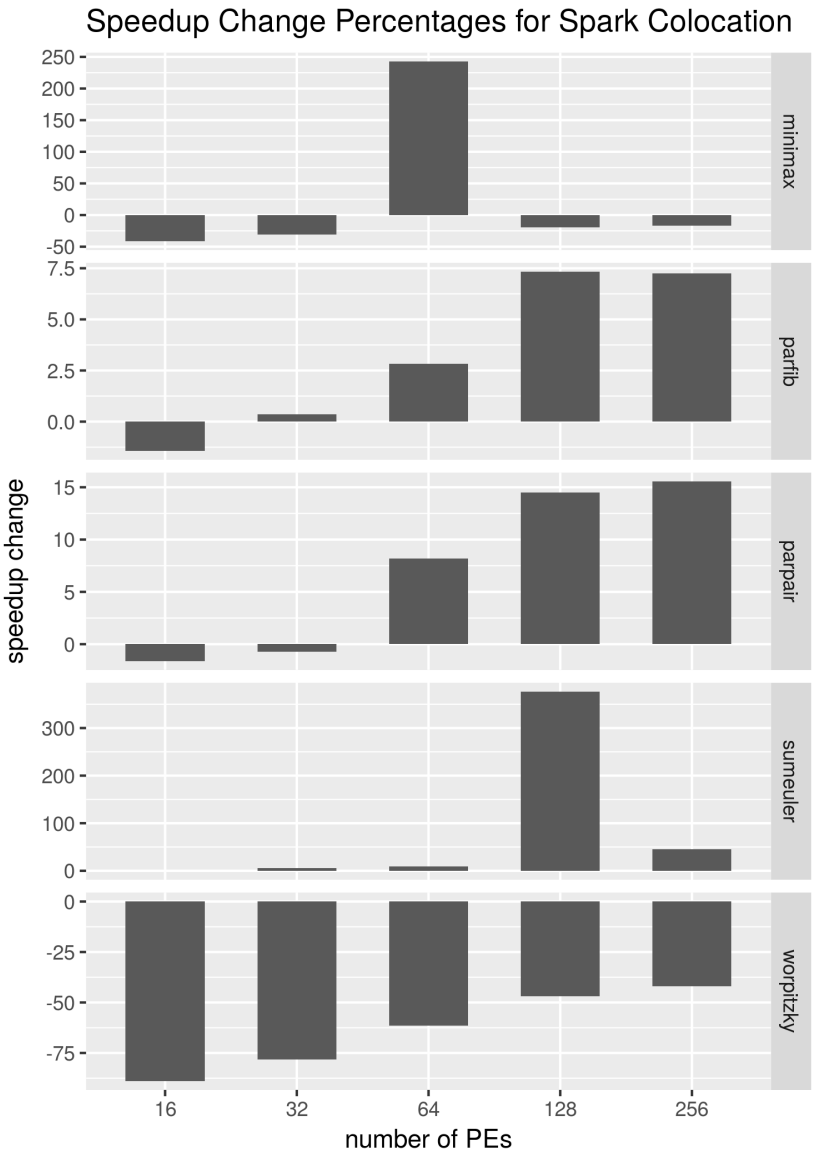


Figure 6.4: Speedup Change in % for SC (higher is better)

The per-application speedup comparison reveals that `parfib`, `parpair`, as well as `sumeuler`, scale well initially and gradually less well for higher PE numbers, whilst `worpitzky` and `minimax` have relatively flat speedup curves. The individual figures show the best performance, whilst summary figures show the median across runs. This way we can observe both, the picture of what is achievable as well as the average trend.

Figure 6.6 examines the speedups for `parfib` in more detail. We observe almost linear speedup for both the default policy and for Spark Colocation on up to 96 PEs. Overall, in all cases Spark Colocation outperforms the default policy with *increased differences for growing numbers of PEs*. The fixed amount of work results in less

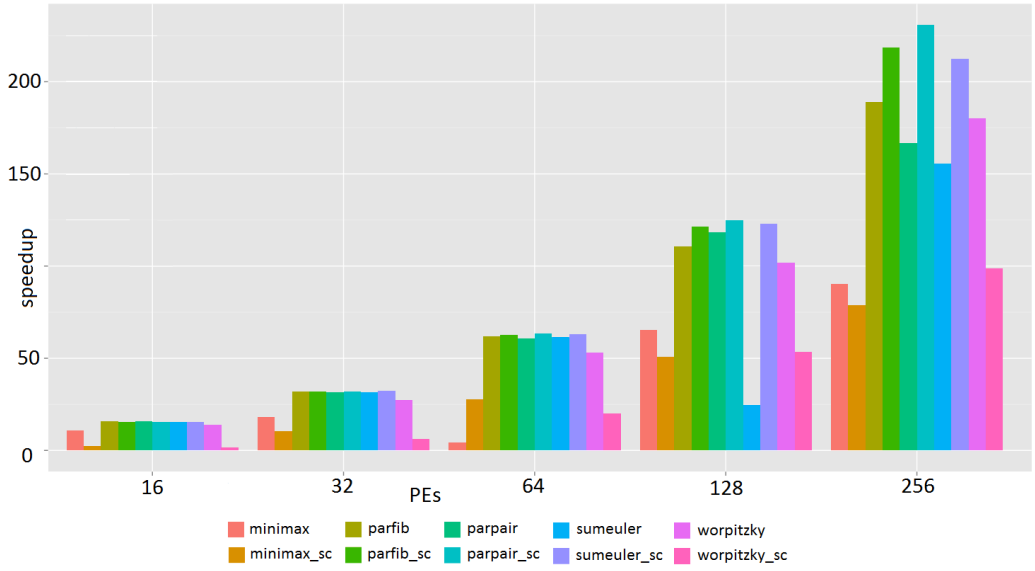


Figure 6.5: Spark Colocation: Speedups

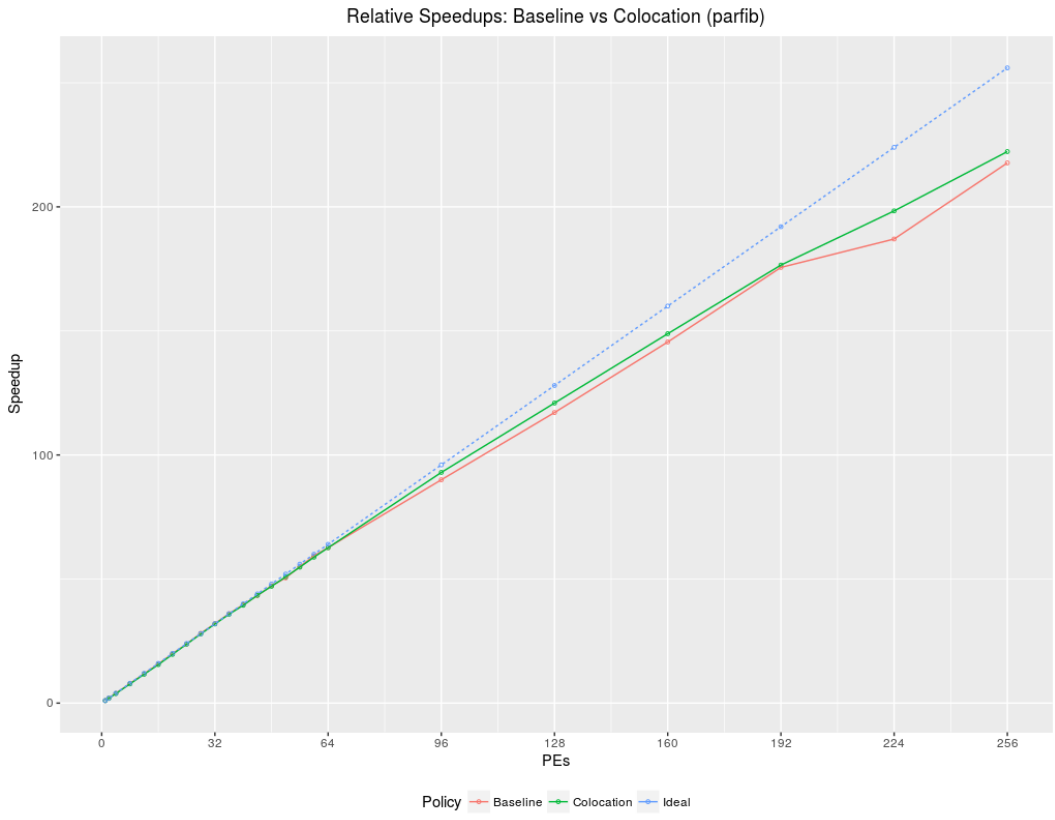


Figure 6.6: Spark Colocation: `parfib` Speedups

steep speedup curves for higher numbers of PEs compared to ideal.

Figure 6.7 depicts the speedup behaviour for `parpair`, where SC again dominates, with both policies exhibiting good scaling of over 200×. However, despite initial linear scaling, we observe earlier flattening out for the default policy from 64

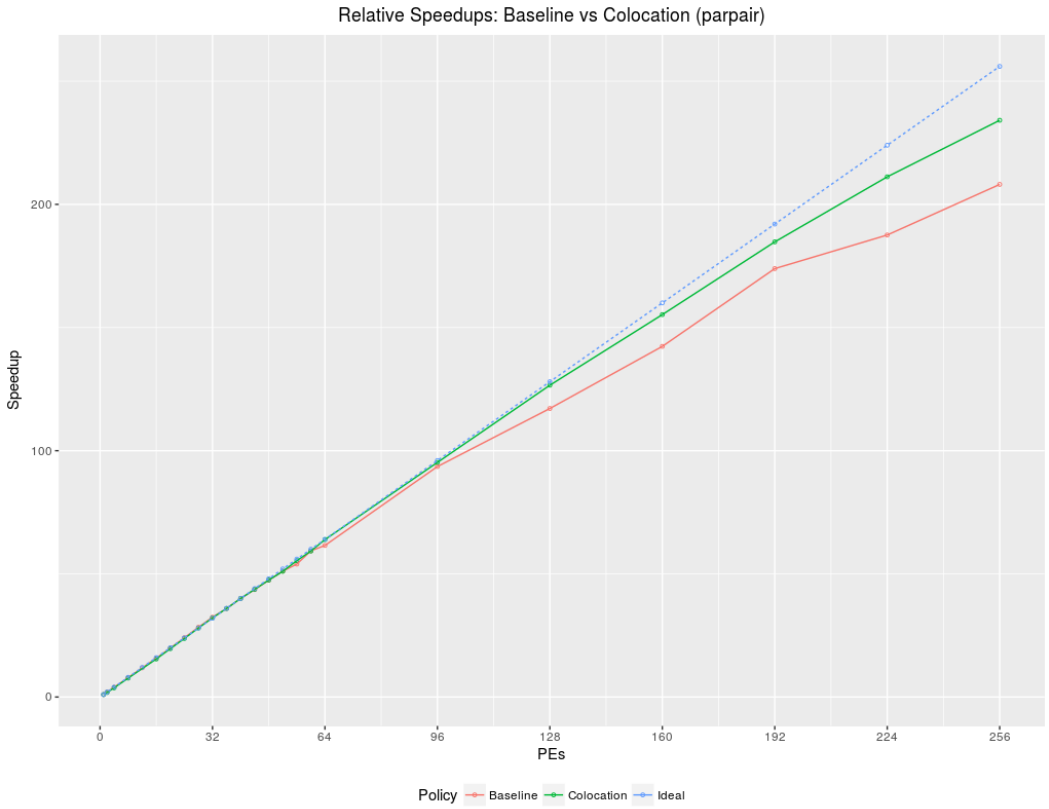


Figure 6.7: Spark Colocation: `parpair` Speedups

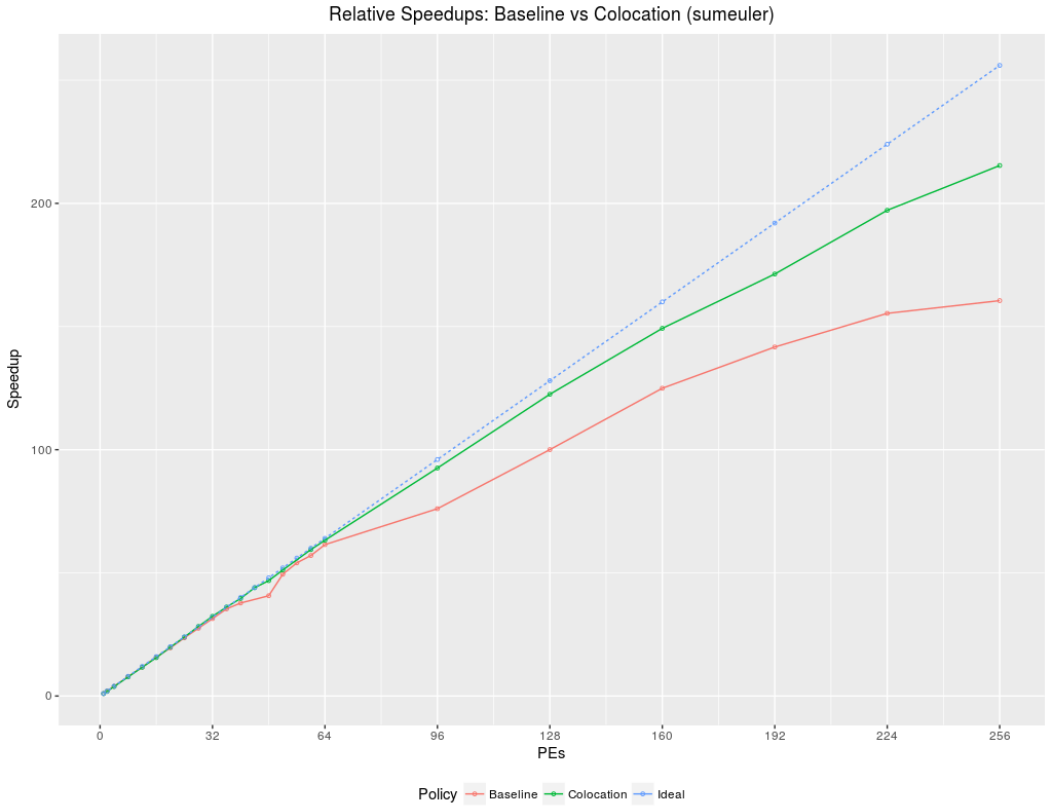


Figure 6.8: Spark Colocation: `sumeuler` Speedups

PEs on. Using SC results in a steeper speedup curve for this nested application.

The situation is similar for `sumeuler` as shown in Figure 6.8. We observe that Spark Colocation appears beneficial in most cases with an increasing gap between the speedup curves for higher PE numbers. Note that inner sparks for this application have similar granularity across spark of the same level, decreasing with the level of the computational tree.

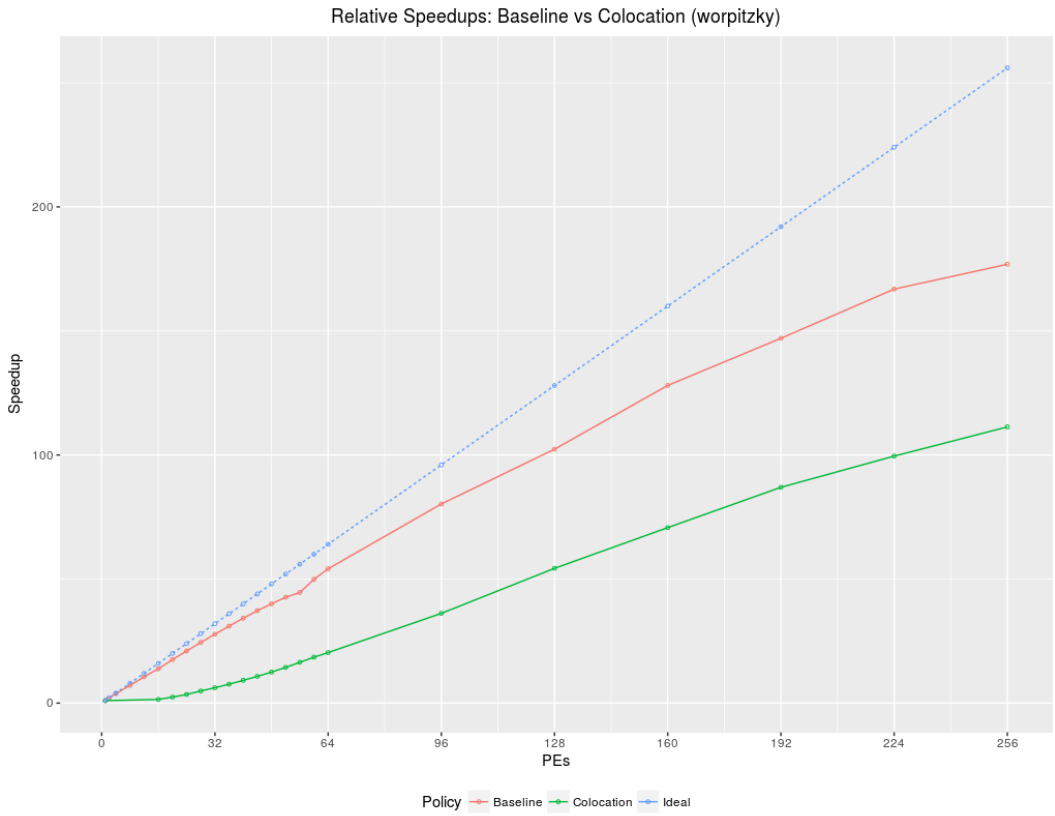


Figure 6.9: Spark Colocation: `worpitzky` Speedups

Figure 6.9 shows a very different picture for `worpitzky`, where the baseline policy consistently exhibits higher speedup than Spark Colocation, which shows a fairly flat initial speedup curve. It fails to catch up with the baseline speedup that is initially close to linear. We suspect that a better threshold setting can be obtained for this application.

An even poorer speedup is reached for both policies for `minimax` as demonstrated in Figure 6.10. The baseline algorithm reaches slightly higher speedups than Spark Colocation, with flattening occurring early for both.

As we don't scale the input size with a growing number of PEs, it is expected

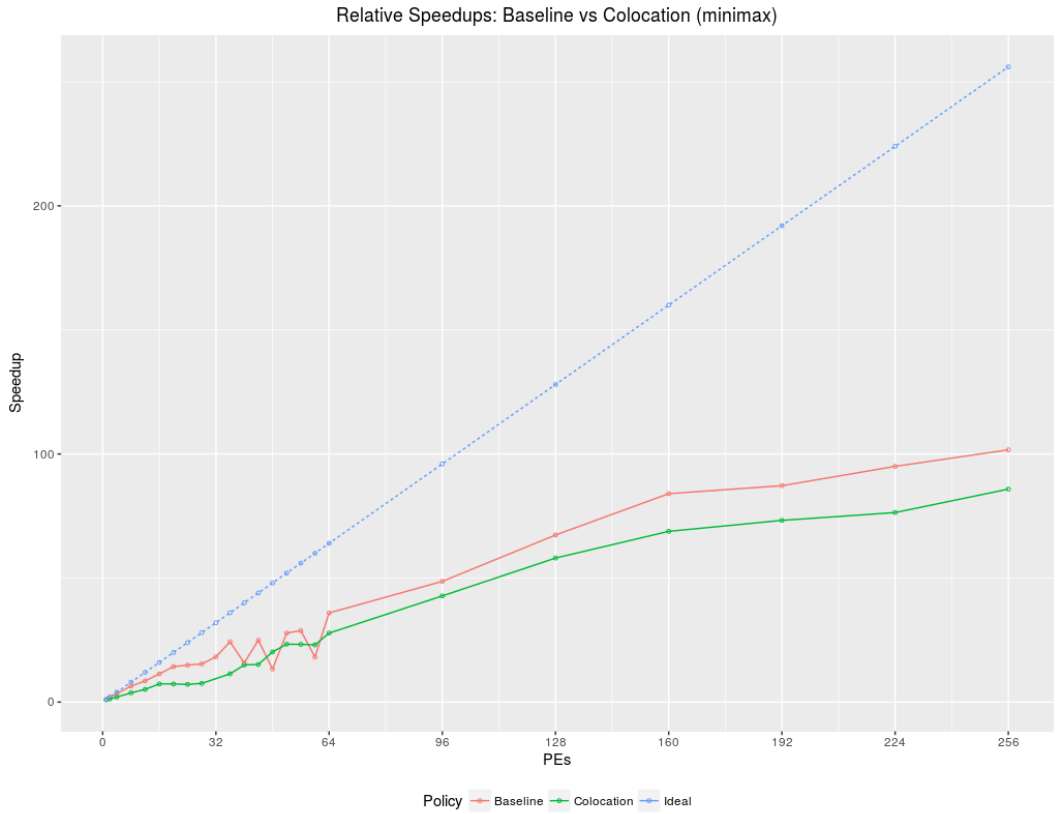


Figure 6.10: Spark Colocation: `minimax` Speedups

that the speedup curves would eventually flatten out due to the limited amount of available work. Otherwise, a speedup curve that is flat even for a small number of PEs suggests that the implementation of the application itself is less scalable.

Next we look at the results from profiling the runs which will help explain the exhibited performance. First, we focus on per-PE load balance over time and on the degree of parallelism, then we examine thread granularity, the time spent fetching remote data and investigate the numbers of transmitted `FISH` messages. Finally, we review the numbers of inter-PE references.

Load Balancing We use event-based profiling to examine thread activity across PEs over time as a measure of utilisation, to compare load balance for SC against the baseline mechanism. We expect a better load distribution for a larger number of smaller threads of less variable granularity, as this allows more flexibility and helps avoid pathological cases which are likely if some threads are disproportionally coarse-grained. Moreover, if SC is effective, we should observe lower times to fetch

needed data as discussed in detail below (see the *Fetching Behaviour* paragraph).

Figures 6.11 – 6.18 provide a picture of the load balance across PEs. They depict the per-PE thread pool sizes over execution time based on the event time stamps for selected median runs on 128 PEs. A deeper shade of green represents a larger number of runnable threads (higher load), whilst red and white gaps show blocking, and blue lines at the bottom of each PE stripe indicate fetching. Unfortunately, no data are available for `minimax` due to instability.

SC mostly results in better load balance due to higher number of active threads across PEs, as visualised by a more even shade of green lines across PEs. We observe reduced idle and blocking time for SC, visualised in red and white.

Moreover, we notice a decreased difference in run time for each PE for SC as compared against the baseline. The execution times themselves are different for the 128 PE case[3] in favour of SC for three out of five applications. These differences are most pronounced for high PE numbers.

These suggest that the RTS is able to exploit the increased parallelism and decreased granularity that result from the use of SC and can better utilise all the available PEs. Fetching occurs at similar times for SC and the baseline because all but the main PE start off idle. More PEs start fetching early at those times and the fetching times are lower for SC, as summarised in Table 6.5, whilst the overall number of fetches is lower for the baseline.

Hence, we argue that, despite the larger number of smaller threads, Spark Colocation can improve load balance by facilitating the sharing of related work which results in fetching of more useful data. Except for `worpitzky` with an excessive degree of parallelism. Next we examine degree of parallelism and thread granularity.

Degree of Parallelism Tables 6.3 and 6.4 demonstrate the measured spark and thread counts over all PEs representing the *potential* and *actual* degree of parallelism, respectively. We report the total counts, the medians and standard deviations across all PEs from the median run profiled on 256 PEs for each benchmark application, comparing the baseline against Spark Colocation.

[3]we have chosen this case because it is the highest number for which visualisation is still readable

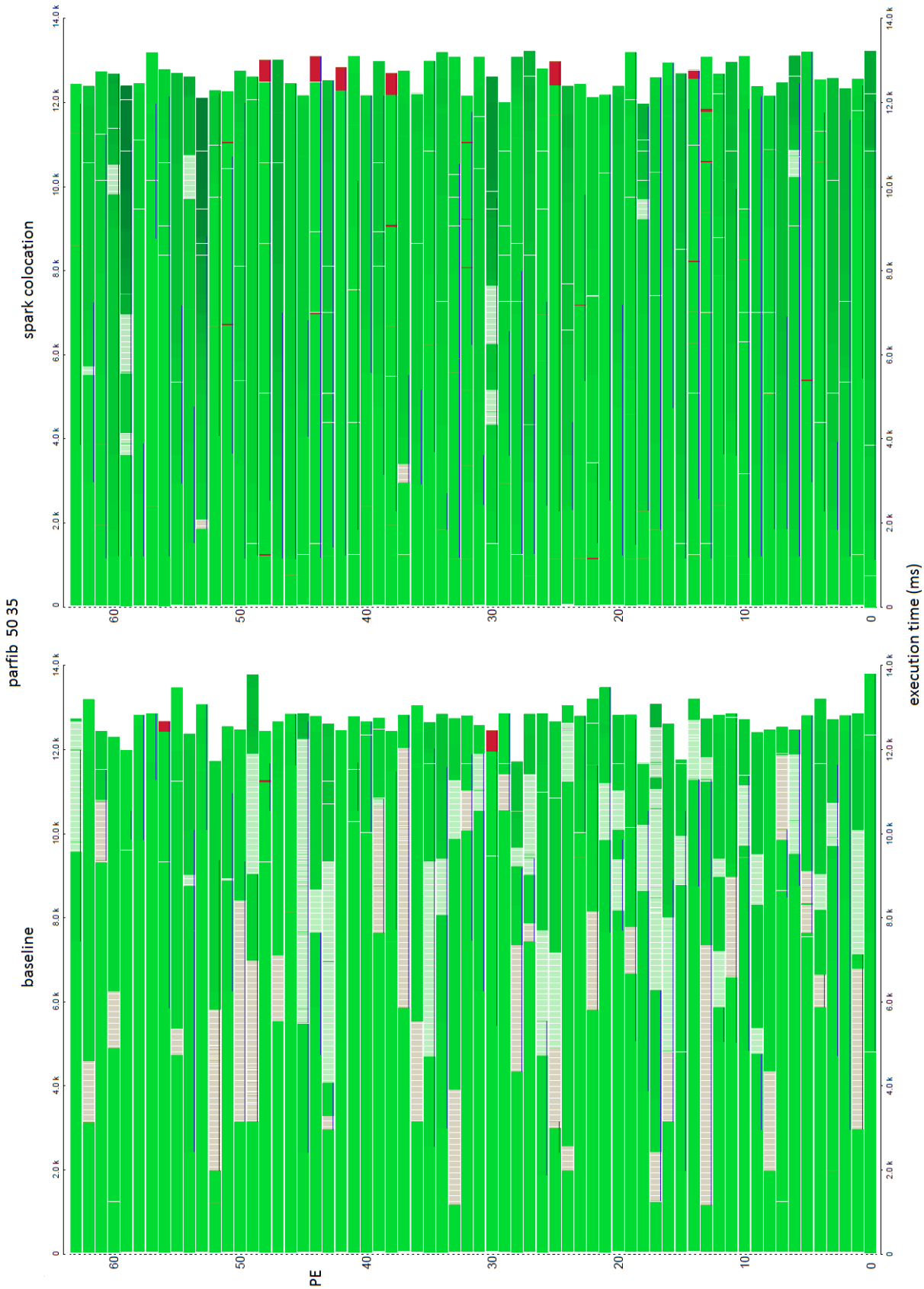


Figure 6.11: Event-Based Load Balancing Per-PE Profile Comparison for `parfib` PEs 1-64 out of 128

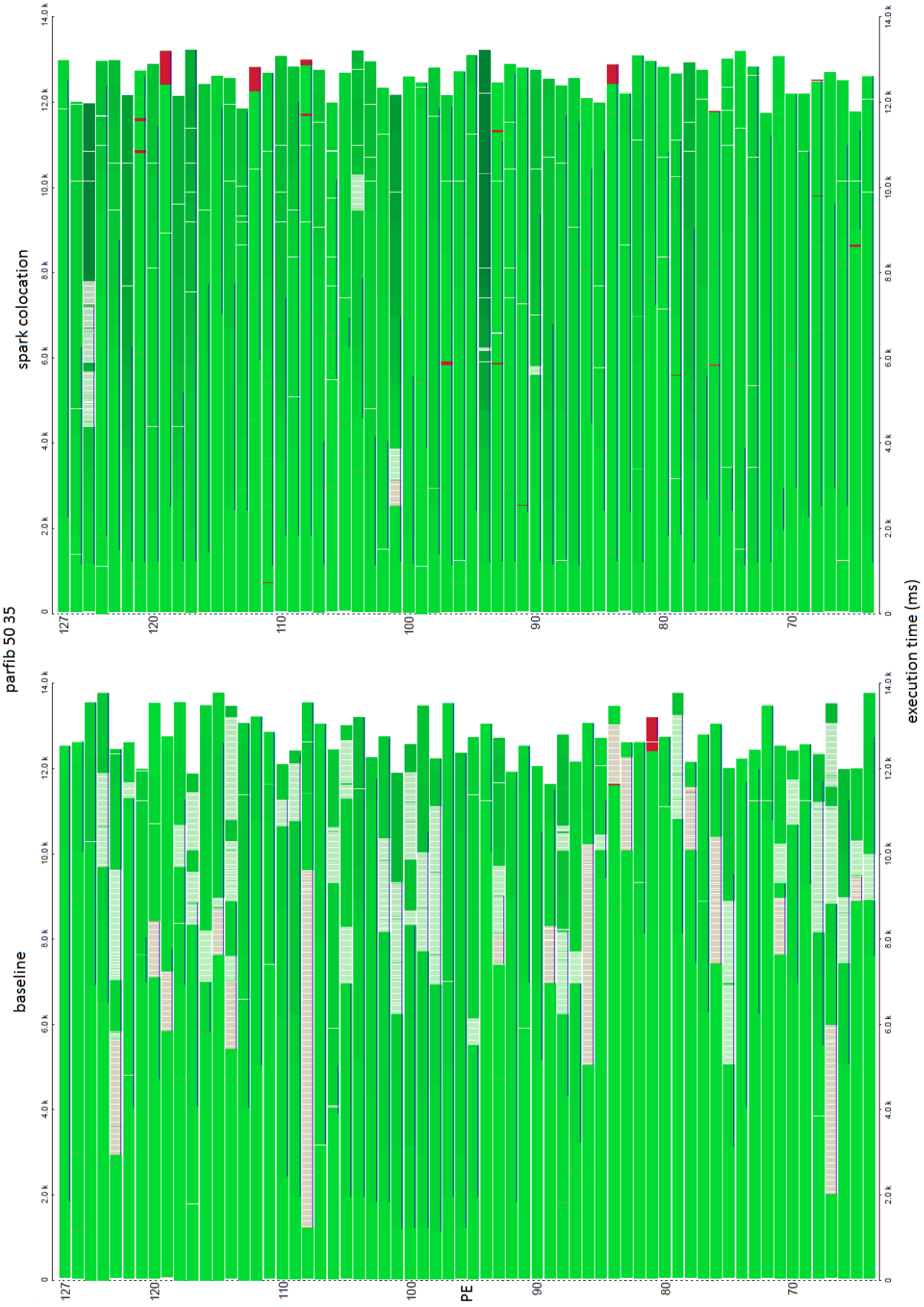


Figure 6.12: Event-Based Load Balancing Per-PE Profile Comparison for `parfib` PEs 65-128 out of 128

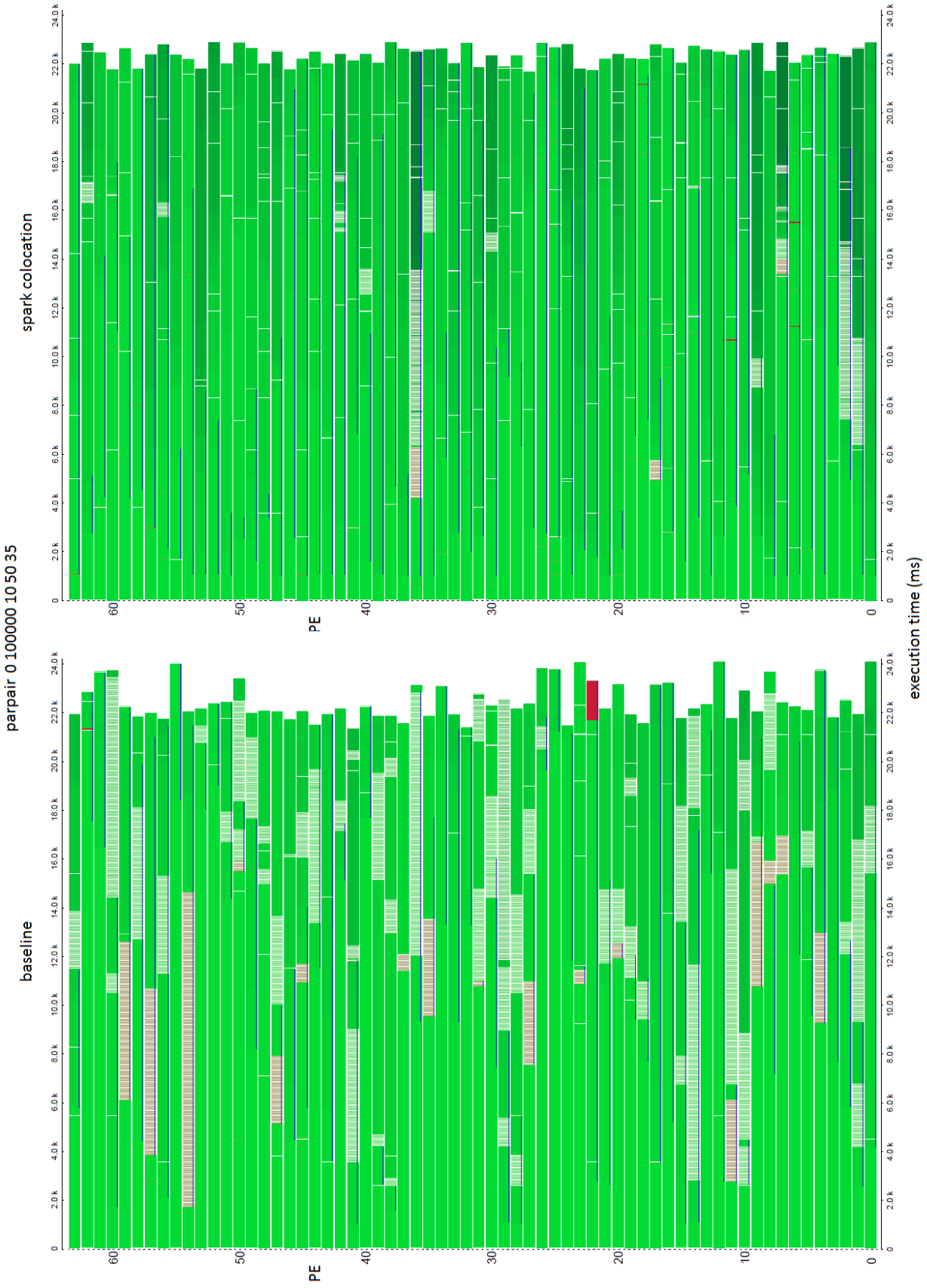


Figure 6.13: Event-Based Load Balancing Per-PE Profile Comparison for `parpair` PEs 1-64 out of 128

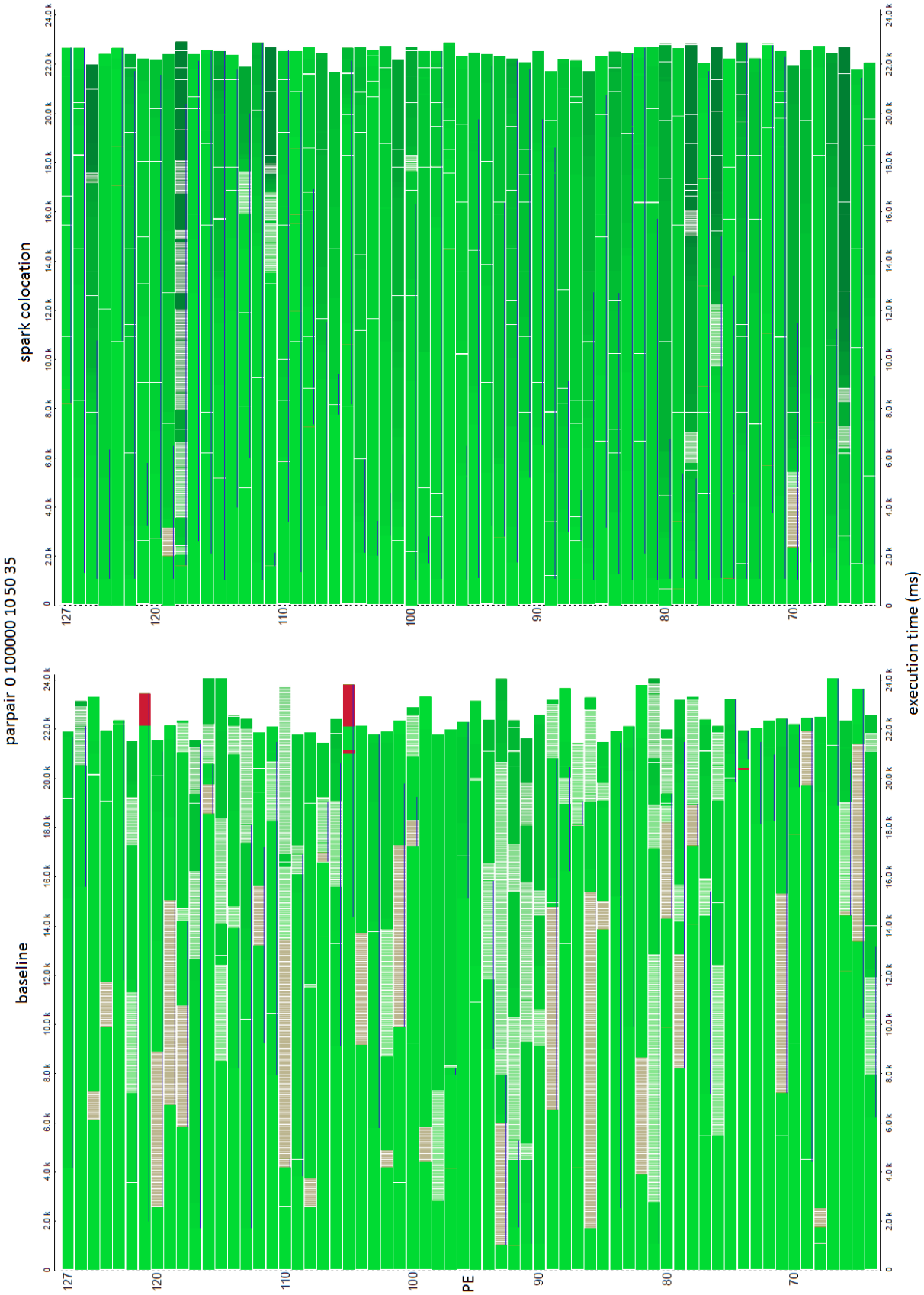


Figure 6.14: Event-Based Load Balancing Per-PE Profile Comparison for `parpair` PEs 65-128 out of 128

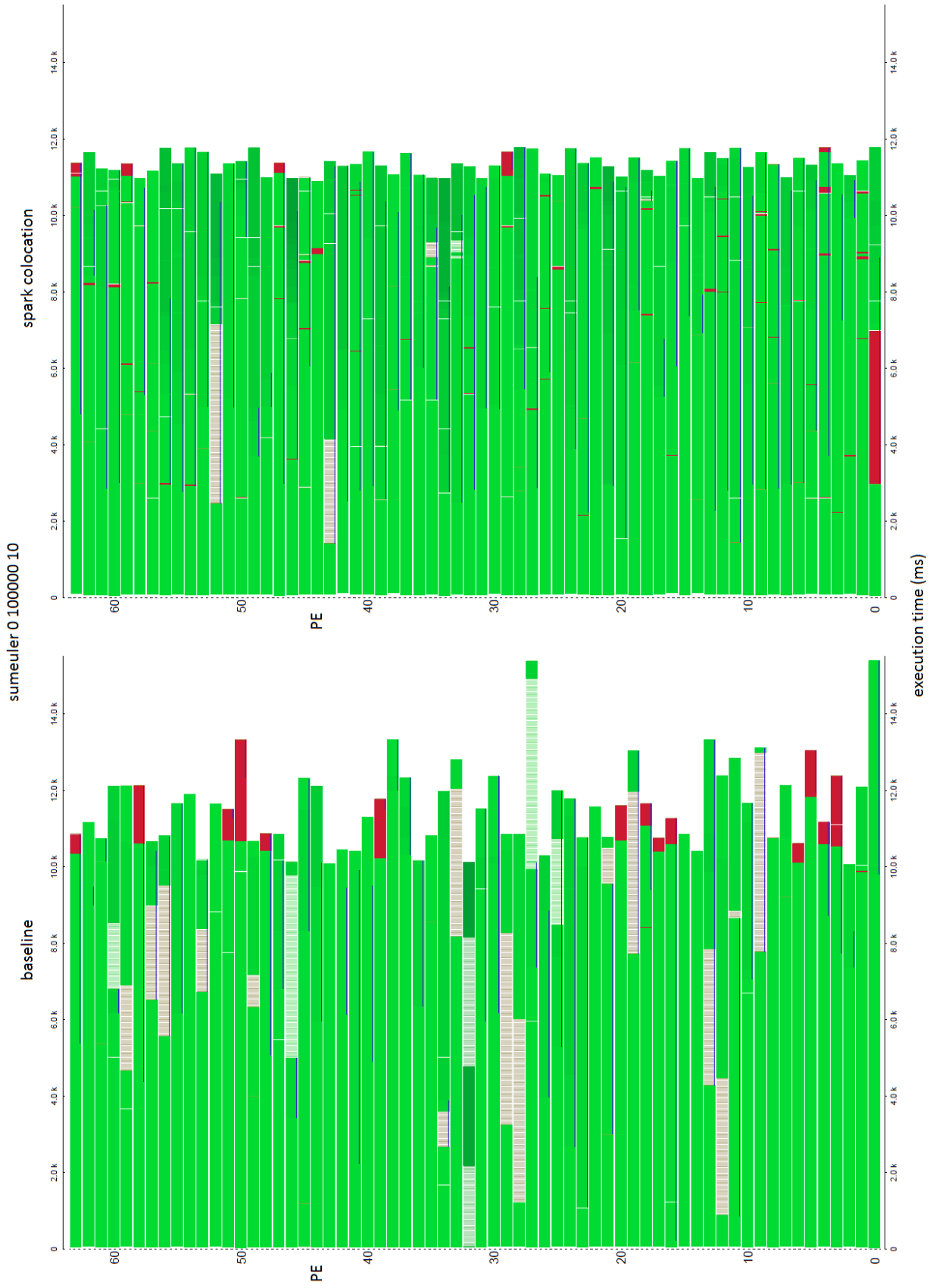


Figure 6.15: Event-Based Load Balancing Per-PE Profile Comparison for `sumeuler` PEs 1-64 out of 128

Figure 6.16: Event-Based Load Balancing Per-PE Profile Comparison for `sumeuler` PEs 65-128 out of 128

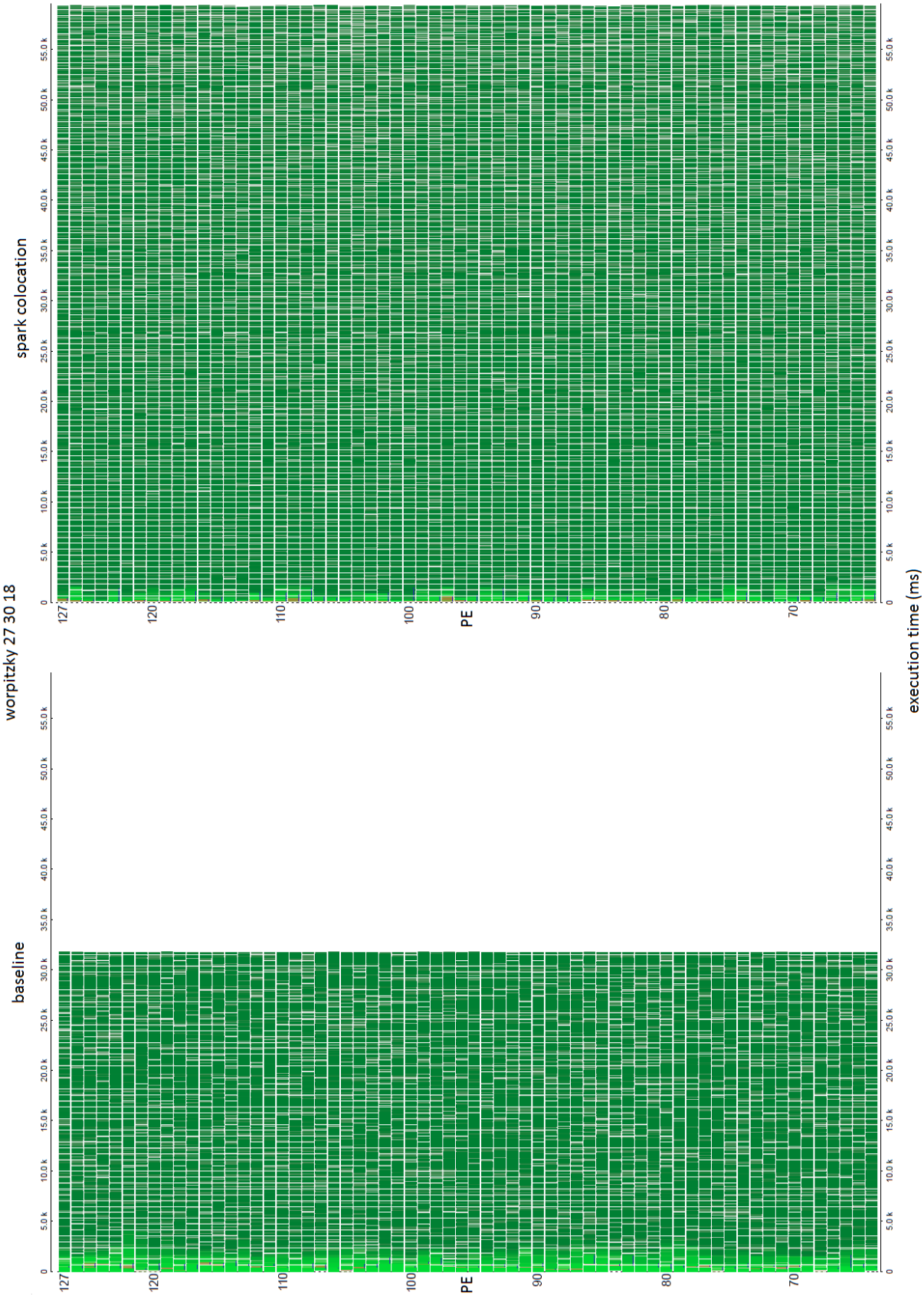


Figure 6.17: Event-Based Load Balancing Per-PE Profile Comparison for `worpitzky` PEs 1-64 out of 128

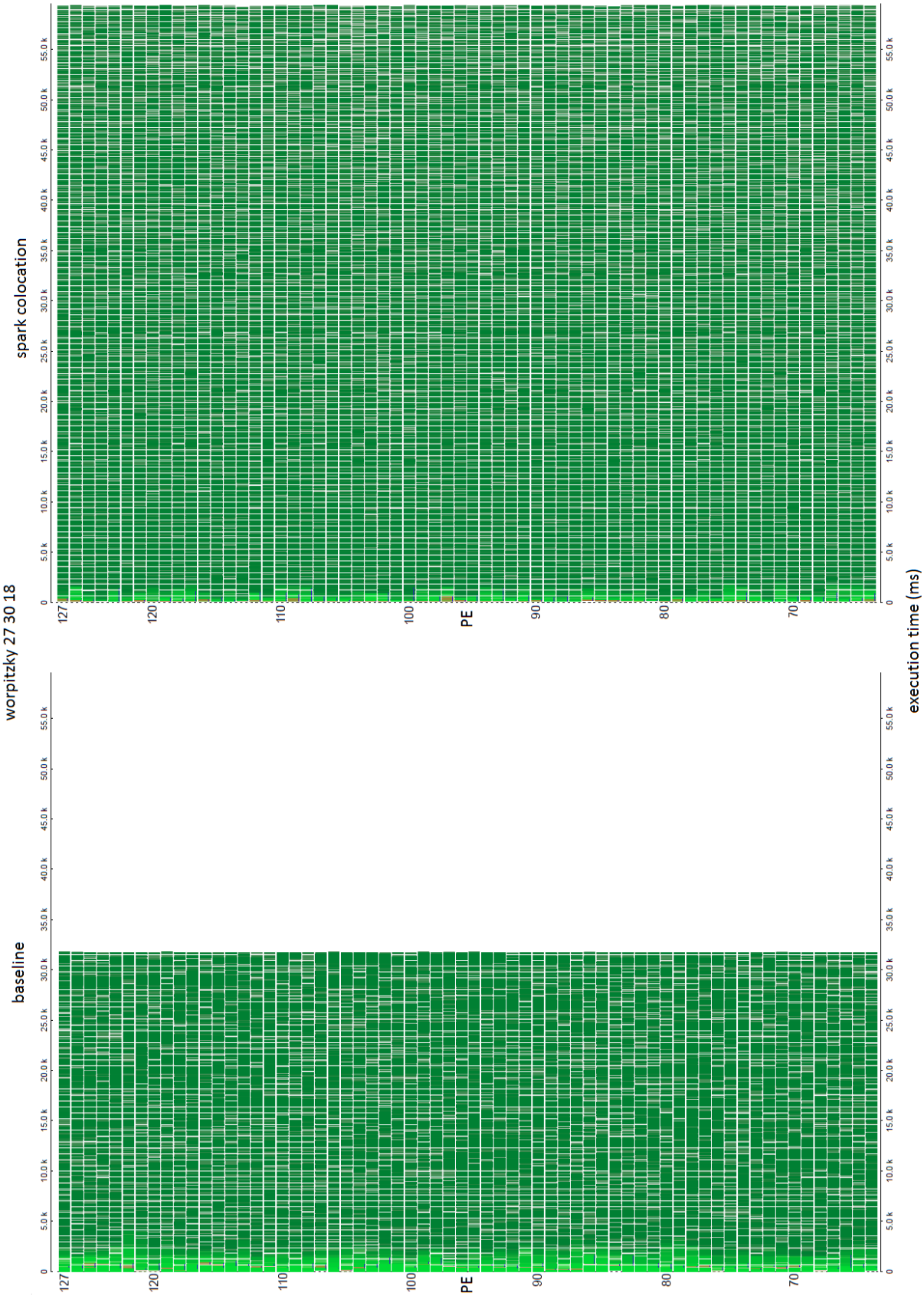


Figure 6.18: Event-Based Load Balancing Per-PE Profile Comparison for `worpitzky` PEs 65-128 out of 128

Table 6.3: Spark Counts for Benchmarks on 256 PEs

application	baseline			SC			**change**
	total	median	stddev	total	median	stddev	**in %**
`parfib`	2755	11	2.28	3172	12	3.46	+15
`parpair`	3840	14	4.34	5045	19	6.24	+31
`sumeuler`	1854	6	3.51	1983	7	4.67	+7
`worpitzky`	337116	1322	88.71	488550	1927	161.14	+45
`minimax`	2466	7	6.31	2525	5	9.90	+2

Table 6.4: Thread Counts for Benchmarks on 256 PEs

application	baseline			SC			**change**
	total	median	stddev	total	median	stddev	**in %**
`parfib`	1127	4	1.02	1584	6	0.64	+41
`parpair`	1195	5	1.35	2508	10	1.54	+110
`sumeuler`	802	3	0.71	955	4	0.97	+19
`worpitzky`	82065	322	31.39	243709	979	82.79	+197
`minimax`	1092	4	1.27	1055	4	1.18	−3

Overall, we observe consistently higher potential parallelism for SC, which can be attributed to the export of related sparks rather then strictly the oldest, which may reduce potential for subsumption once the computation is shared across the PEs. This turns out to be particularly beneficial for larger numbers of PEs as the number of threads per PE is increased in all but one case (`minimax`). The `worpitzky` benchmark shows that although beneficial for load balancing, having a higher number of threads may become counterproductive when there are already more than enough threads in the baseline case due to additional overhead.

Moreover, we notice the larger number of converted threads that represent actual parallelism for SC, except for `minimax`, which exhibits little change (see Table 6.4). This results from the higher number of created sparks (see Table 6.3) and is useful for a higher number of PEs to spread the load across more nodes and PEs, potentially reducing idleness. The `worpitzky` program exhibits overwhelming parallelism management overhead leading to poor scalability and is an example of worst-case behaviour. Both spark and thread counts per PE in Figures 6.19, 6.20 and 6.21 show that most of the 256 PEs complete work with each using multiple threads.

The spark counts suggest that the differences between the mechanisms are rela-

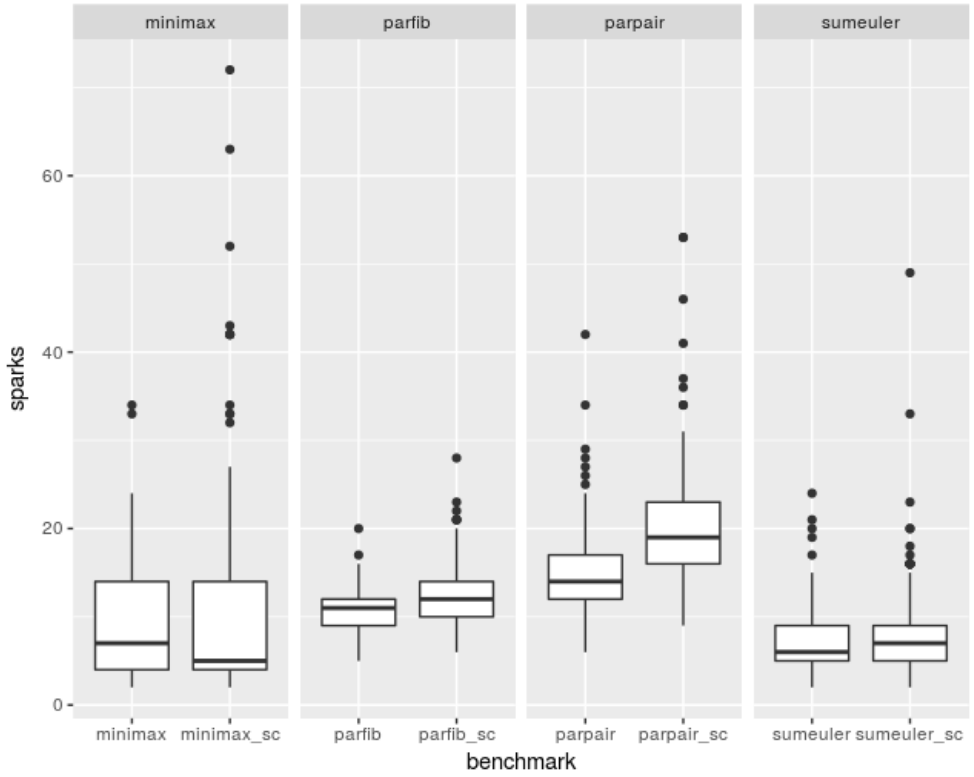


Figure 6.19: Sparks per PE on 256 PEs

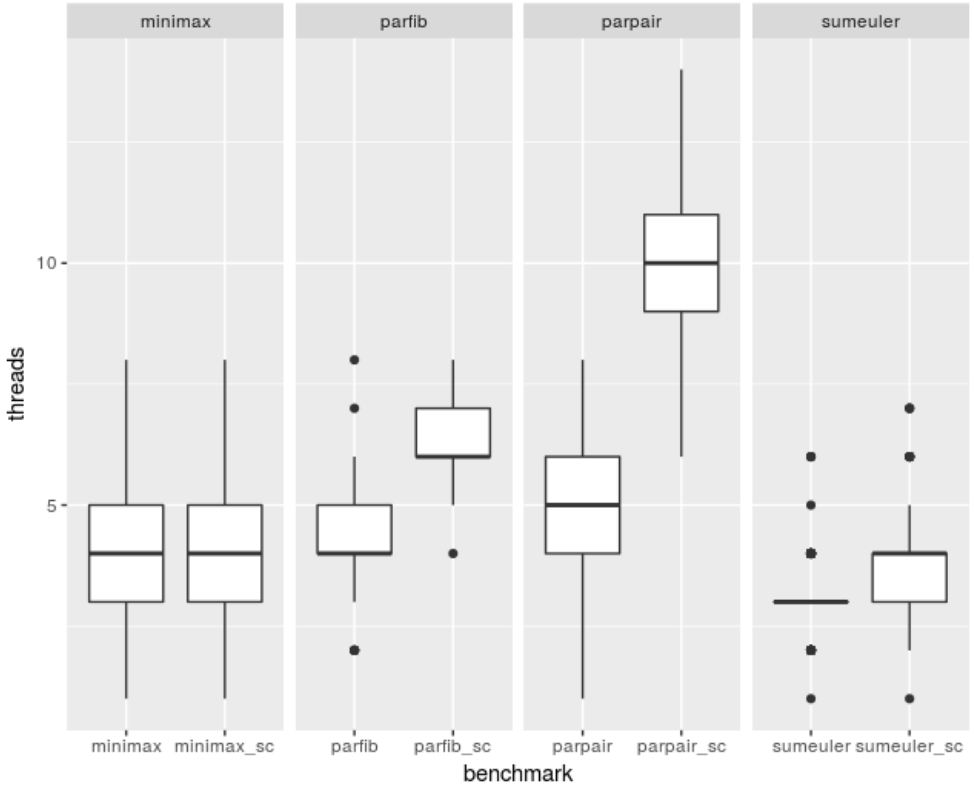


Figure 6.20: Threads per PE on 256 PEs

tively low, with slightly higher potential parallelism for benchmarks for which Spark Colocation shows an improvement, whereas it is lower otherwise. An exception is `worpitzky`, which exhibits much higher potential and actual parallelism for SC but the performance and scalability are negatively impacted by the additional overhead of managing excessive parallelism.

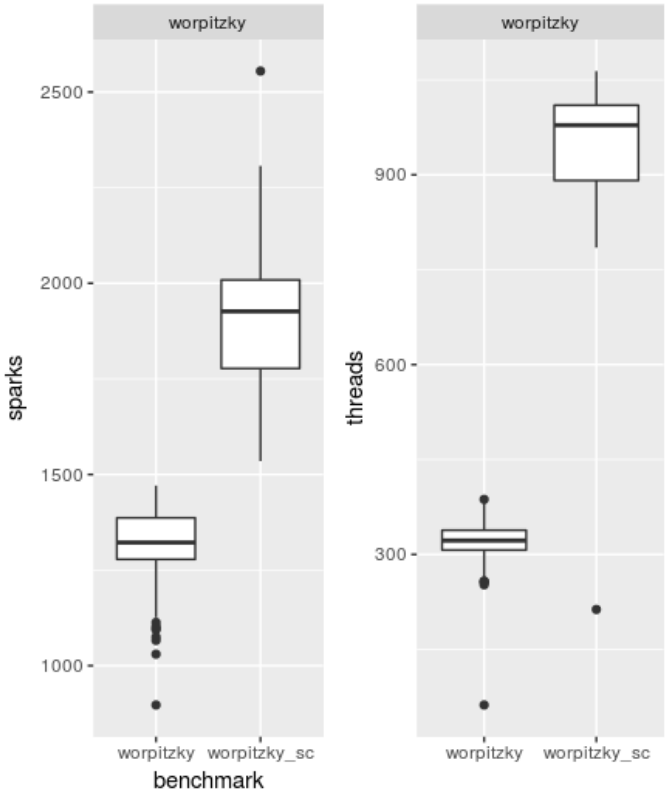


Figure 6.21: Sparks and Threads per PE on 256 for `worpitzky`

The thread figures show a distinction between benchmarks that perform better with more threads with SC and the benchmarks with no difference. This summary is complemented by the per-PE load balancing data, emphasizing the desirability of spreading the load evenly across PEs, spatially as well as temporally.

However, the granularity of threads is not visible from this display, so that even though all the PEs have enough sparks to convert some of them into threads, some of the sparks and associated threads may be relatively small whilst others require more computation. In particular, if one PE receives a disproportionally larger task in the end of the execution, others may run out of work and go idle.

Granularity We compare the distribution of thread granularities exhibited by a program, representing thread sizes in terms of run time. Ideally, a program would be composed of equal-sized independent computations which would make parallelisation relatively easy. The granularity should be at least larger than the thread creation overhead. In practice, most tasks are of different sizes and lead to more complex work distribution decisions, making it hard to obtain optimal solutions. GUM's profiling sub-system was extended to provide the per-thread granularity information (see Appendix B.4.2 for details).

The granularity profiles in Figures 6.22 – 6.25 show that Spark Colocation consistently generates *more threads* of *smaller granularity*, offering more opportunities for load balancing for higher PE numbers, but also increasing overhead. We focus on the data for 256 PEs as the difference in run time is most pronounced in those cases. Note the difference in thread numbers compared to Table 6.4, which is due to work-stealing non-determinism, because we use data from a different run with event-based profiling turned on.

Figure 6.22 shows the granularity distribution for `parfib` where Spark Colocation results in more short-lived threads than the baseline. The threads for SC are clustered around the 1000ms mark, whereas the granularity is less even for the baseline case ranging from 500 to 5000ms. This supports the intuition that applications with a larger granularity range are more sensitive to scheduling decisions as allocating larger computations to the same PE could lead to load imbalance.

On the other hand, PEs with larger threads are unlikely to be actively looking for more work unless using the watermark mechanism. However, they may still be interrupted by the `FISH` messages, which they would forward on unless they are generators of parallelism, which is unlikely in the initial phases of D&C computations.

We see a similar picture for `parpair` in Figure 6.23 (note the different scales across granularity figures). The peaks are located in the same buckets in the histogram showing that SC does not fundamentally change application's granularity profile. We rewrote the originally data parallel `sumeuler` in D&C style to take advantage of SC[4], so it exhibits behaviour alike to the other applications.

[4]in a flat data parallel program all sparks can be considered direct siblings at the same level

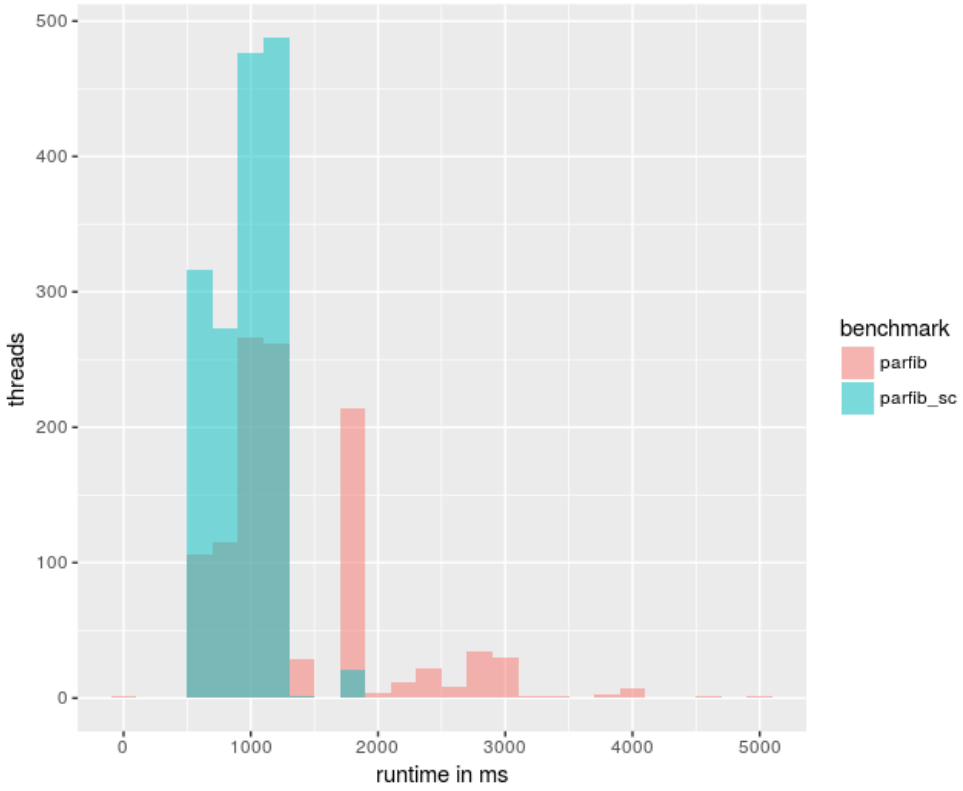


Figure 6.22: Granularity of `parfib` on 256 PEs

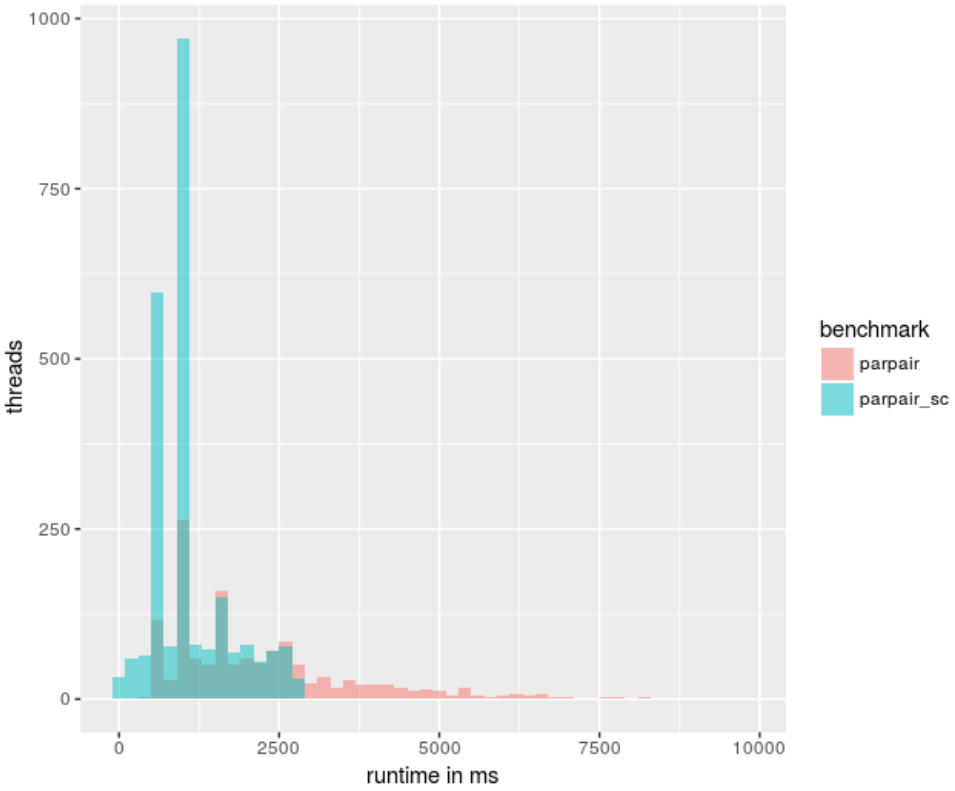


Figure 6.23: Granularity of `parpair` on 256 PEs

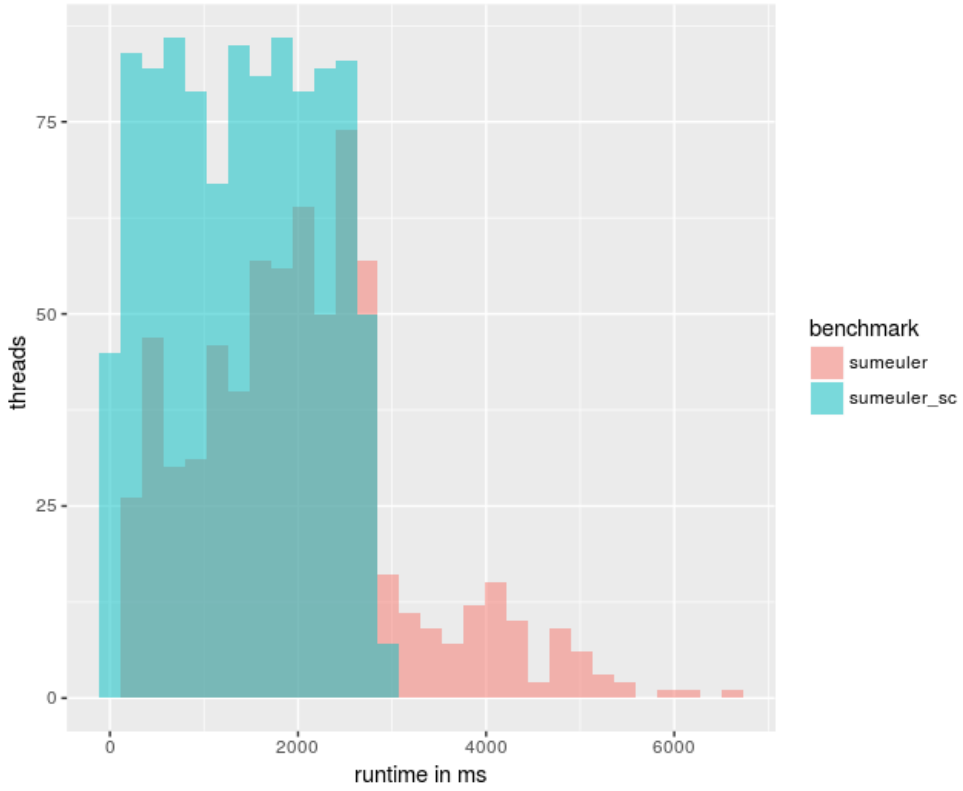


Figure 6.24: Granularity of `sumeuler` on 256 PEs

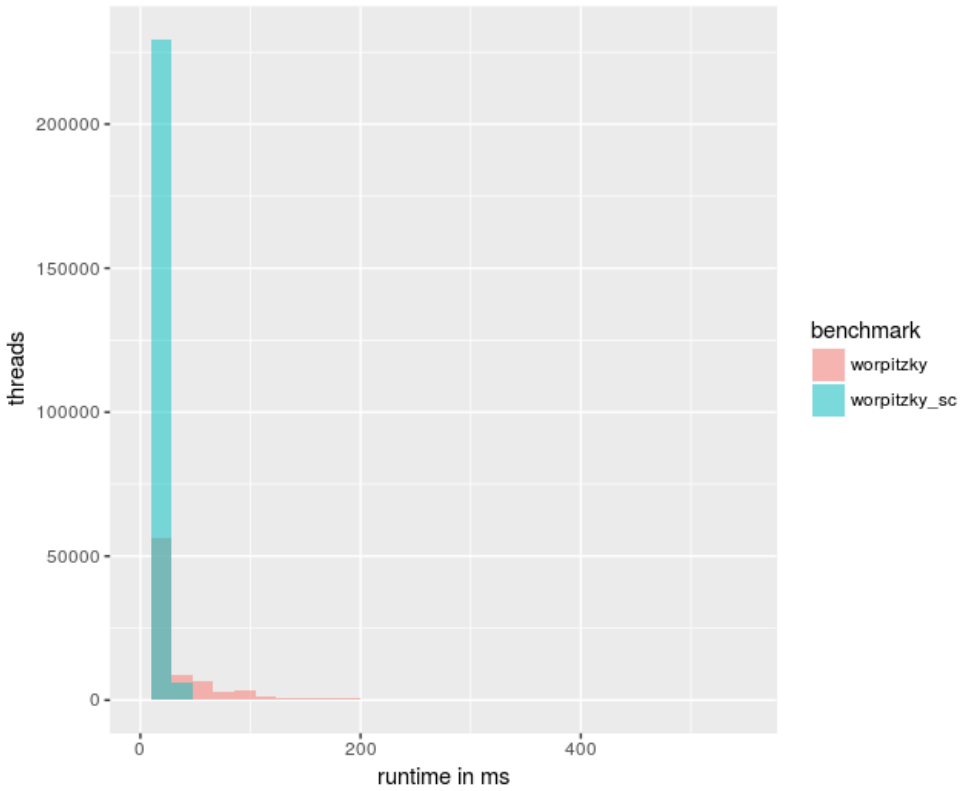


Figure 6.25: Granularity of `worpitzky` on 256 PEs

An extreme case is `worpitzky` where a lot of tiny threads (well over 225000 threads of around 20ms granularity) are generated which negatively impacts scalability. We have kept the parameters for SC and baseline the same for comparability, but it appears that `worpitzky` with SC may benefit from a different threshold setting to limit the amount of threads[5].

Fetching Behaviour Another distinguishing characteristic is the *fetch time* threads spend waiting for data required by the computation to arrive. Table 6.5 compares the baseline and SC across applications for the median run on 256 PEs.

Table 6.5: Summary of Fetching Behaviour on 256 PEs

application	baseline mean (ms) fetch time across PEs	colocation mean (ms) fetch time across PEs	**mean fetch time change in % across PEs**	total fetch time change in %	total fetch count change in %
`parfib`	829.24	637.11	−23	+8	+35
`parpair`	1109.11	565.49	−49	−5	+78
`sumeuler`	593.84	290.17	−51	−29	+49
`worpitzky`	19.02	12.19	−40	+81	+163

In some cases it is possible that the data is already available or fits into the same packet, resulting in fetch time of zero, as for many `sumeuler` threads, and in other cases the fetch time may exceed the time the thread spends performing the computation. We observe that *SC has consistently a smaller mean fetch time across PEs* than the baseline with decrease in the range between 23% and 51%. This suggests that SC is indeed effective, by indicating that the threads in SC case are 'more useful' in the sense that they spend less time waiting on data to arrive, which is what SC design aims to achieve. Thus, despite smaller granularity, SC threads have higher average utilisation as can be seen from the load balancing results, and the degree of parallelism is increased.

Additionally, the total number of fetches is increased due to the larger number of threads, but in most cases both the mean number of fetches per PE and the standard

[5]unfortunately the other input parameters we tested lead either to too short or too long sequential run time

deviation are slightly lower for SC. Next we examine communication characteristics as another source of overhead.

Communication Figure 6.26 shows the number of the `FISH` messages across the applications, which is an indicator of the need to obtain remote work, and compares Spark Colocation against the baseline mechanism. Note the logarithmic scale used due to a large difference in numbers across applications. As expected, we see an increase of `FISH` counts with increasing number of PEs.

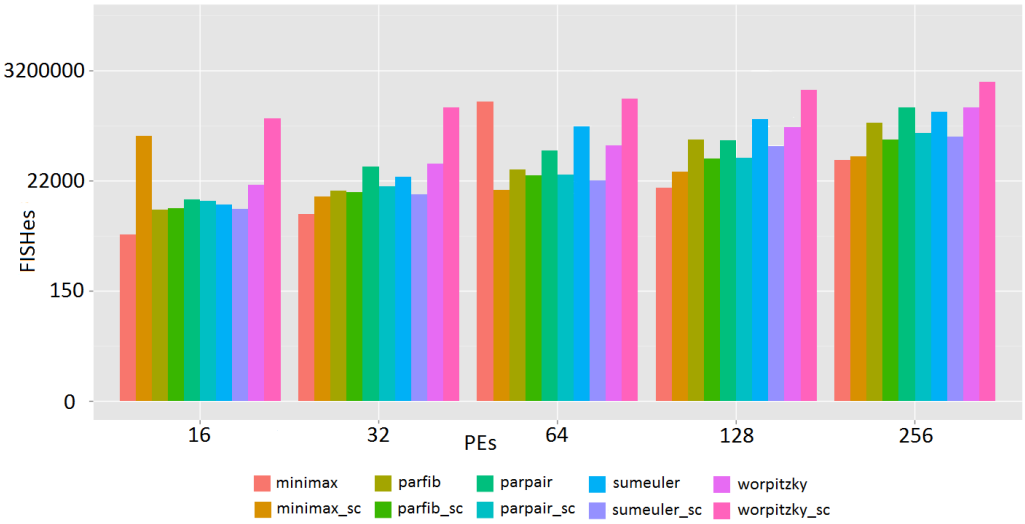


Figure 6.26: Spark Colocation: `FISH` Message Counts (log scale)

We also observe *lower* `FISH` counts *for SC for cases in which the new mechanism outperforms the baseline.* This is due to improved load balancing leading to less idle PEs and it is an improvement as it leads to reduction in communication costs by decreasing the number of messages sent.

Global Indirection Table Residency Figure 6.27 depicts the median number of global indirections across PEs for baseline runs and runs with SC on up to 256 PEs which shows the degree of sharing across PEs using the distributed shared heap.

We notice that the number of global addresses is consistently higher for SC suggesting an increase in the amount of inter-PE sharing. This is due to the higher parallelism degree and the sharing pattern that favours sharing related sparks rather than oldest. In Table 6.5 we have observed a reduced average fetching time despite

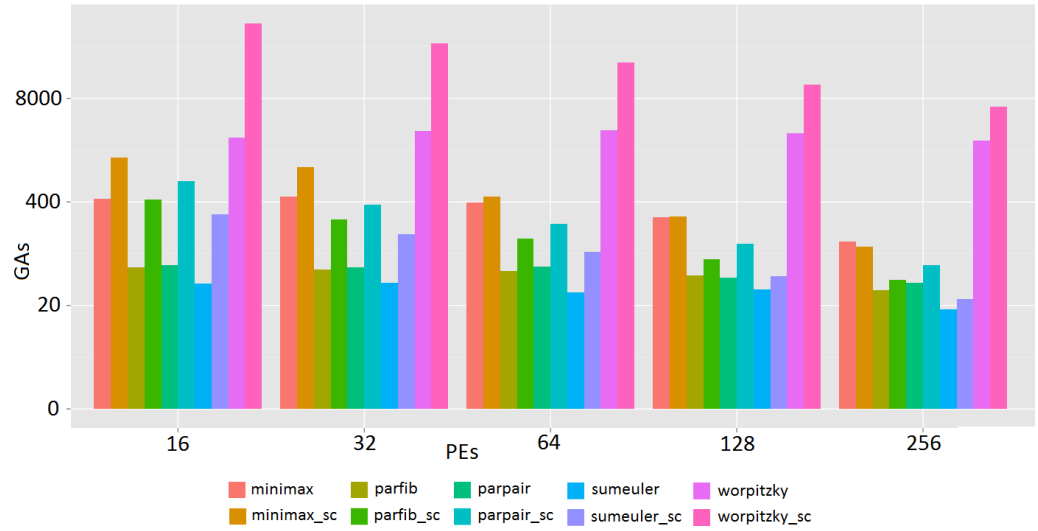


Figure 6.27: Spark Colocation: Median Global Addresses (log scale)

more global addresses as more useful data is exchanged. In particular, we observe an excessive number of global addresses for `worpitzky` where the parallelism degree is very high.

An interesting pattern is that the difference consistently decreases with an increasing number of PEs. This suggests that SC scales better with the number of PEs and may be able to overtake the baseline for even higher PE numbers, whereas the baseline mechanism works well for lower numbers of PEs. We attribute this behaviour to the higher parallelism degree and sharing of smaller sparks that require more fetching messages with shorter average fetching time for SC as opposed to the baseline.

6.4 Discussion

Comparing SC to the baseline mechanism, the results show speedups and speedup improvements of up to 46% with SC on 256 PEs for the three more regular and nested applications: `parfib`, `parpair`, `sumeuler`. For `minimax`, which exhibits stronger irregularity and limited scalability, and for `worpitzky`, which is excessively fine-grained, we observe performance degradation.

When using SC, the distributed graph reducer shares *related* work resulting in a

higher degree of both potential and actual parallelism, and more fine-grained and less variable thread size. In particular, we observe a higher thread conversion increase rate than the increase rate of sparks for SC, which suggests that less threads are subsumed because younger sparks are shared (see Tables 6.3 and 6.4). Having more sparks that are also more fine-grained allows for more flexibility when balancing the load and improves scalability on higher PE numebrs for SC.

We validate this behaviour by observing a reduction in average fetch times of between 23% and 51% for SC, suggesting improved locality. This results in improved load balance for `parfib`, `parpair`, `sumeuler`. This is despite the increased amount of `FETCH` messages and of inter-PE pointers, resulting from larger number of sparks.

As expected, SC appears most beneficial for higher numbers of PEs where improved load balancing and higher degrees of parallelism have more opportunities to pay off. The SC mechanism is therefore effective in improving scalability.

In more general terms, we show that a RTS can beneficially use the ancestry information that was originally lost during compilation. However, a balance needs to be struck with the overhead incurred by the finer thread granularity.

Finally, even though we have placed the annotations manually it is possible to automatically enumerate `par`s, changing them to `parEnc`s. Recent work [50], which complements this work, investigates automated placement of `par` annotations as a step towards fully-implicit parallelism.

Chapter 7

Conclusion

This thesis contributes to the study of efficient distributed execution of semi-explicit parallel non-strict functional languages on distributed-memory architectures. We characterise a set of parallel programs and investigate two novel approaches to improve the work stealing mechanism and compare their performances to the baseline random work stealing scheduler.

7.1 Summary

First, we characterise a set of small and medium-sized parallel functional applications run on a server-class NUMA multi-core and on a cluster of multi-cores using either GUM or SMP RTS. We identify the key sources of coordination overhead in terms of associated metrics such as communication rate, heap and GA residency, allocation rate, and thread granularity. Detailed profiling reveals diverse bottlenecks and helps gain insight into dynamic application behaviour and into RTS-level aspects. In particular, we observe that if either of the chosen metrics is high the performance and scalability are likely to degrade. The results also reveal a strong correlation between GA residency and the amount of communication. Moreover, thread subsumption appears to work well with D&C applications as a way to throttle parallelism. Additionally, we find GUM behaviour similar and consistent across architectures and identify and confirm a scalability bottleneck within SMP. These insights inform the approaches to work stealing we set out to investigate.

The first approach we study uses historical information to adapt the choice of the victim by preferring processors with successful steal attempts in the past. As expected, we observe substantially lower numbers of messages, in particular of `FISH` messages, for data-parallel and nested applications.

However, this heuristic fails in cases where past application behaviour is not resembling future behaviour, for instance for D&C applications with large number of very fine-grained threads and generators of parallelism that move dynamically across PEs. This mechanism is not specific to the language and the RTS, and applies to other work stealing schedulers.

In the second approach, we focus on the other key work stealing decision of which sparks to donate, investigating the effect of Spark Colocation. When using SC, the PEs in the distributed graph reducer donate *related* work as evidenced by reduced fetch times and decomposes the parallelism into smaller work units as suggested by the granularity profiles. This results in a higher degree of both potential and actual parallelism, and more fine-grained and less variable thread sizes, which helps to improve load balance, in particular for higher numbers of PEs. Moreover, reduced fetch times provide evidence for improved locality. We observe reduced run time for three programs: `parfib`, `parpair`, and `sumeuler` despite increased number of `FETCH` messages and of inter-PE pointers for SC.

The results show high speedups both for the baseline and for SC, and speedup improvements of up to 46% with SC on 256 PEs for the three more regular and nested applications out of five, and performance degradation for two programs, one of which is excessively more fine-grained and another one exhibiting limited scalability. Using SC results in higher parallelism degree and more fine-grained threads and the fetch times are consistently reduced by between 23% and 51%, suggesting improved locality, despite the increased amount of inter-PE pointers.

As expected, SC appears most beneficial for higher numbers of PEs where improved load balancing and higher degrees of parallelism have more opportunities to pay off. Therefore SC improves application scalability and we expect better scalability beyond 256 PEs.

In more general terms, we show that a RTS can beneficially use historical information on past stealing successes that is gathered dynamically and used within the same run in most D&C applications, as well as the ancestry information that was originally lost during compilation, but is reconstructed using programmer annotations that are forwarded on to the RTS at run time, in data parallel and nested applications with stable sources of parallelism. Moreover, the results support the view that different heuristics are beneficial for applictions using different parallelism patterns, mandating a flexible approach.

In summary the main contributions are as follows:

- Design, implementation and empirical evaluation of the Spark Colocation mechanism for distribution of advisory parallelism for D&C and nested parallel applications dynamically using the ancestry relation that reflects the proximity of tasks in the compute tree;

- Design, implementation and empirical evaluation of the generic History-Based Work Stealing mechanism which uses recent past stealing success and failure information to select work stealing victim PEs at run time within the current run;

- Characterisation of applications written in a non-strict parallel functional language using ends-based as well as means-based metrics, identifying sources of coordination overhead and confirming the differences between applications with different parallelism patterns;

- Introduction of an adaptivity classification scheme that can be applied to run-time system mechanisms for parallelism management, complemented by a survey of high-level parallel programming models and an overview of langauge run-time systems.

Next we identify several limitations of this work and suggest multiple directions for future work.

7.2 Limitations

We use a limited number of benchmark programs of small-to-medium size, lacking large-scale applications. Moreover, we mainly use a commodity cluster as the hardware platform. Currently, there is no agreement among researchers as to what constitutes a minimal and representative set of applications necessary to assess runtime system performance and scalability.

For History-Based Stealing our study leaves out the investigation of the accuracy–coverage trade-off, which appears important for achieving performance improvements, as if either accuracy or coverage is too low, using historical information may become counterproductive. We have manually selected a reasonable invalidation interval for each application, but we believe a more systematic treatment of the search space would merit a separate project.

For Spark Colocation, we require the programmer to manually annotate the program to identify the sources of parallelism. Although a reasonable experimental trade-off that facilitates prototyping, it makes it more difficult to more widely adopt the use of the method as applications need to be rewritten. We believe a compiler would be able to automatically annotate GpH programs by enumerating the respective sources or parallelism.

Additionally, we have only compared the baseline to SC with maximum prefix matching on encodings, which is only one possible, albeit fitting, way to compare ancestry among sparks. Perhaps the encoding itself could be designed in a different way to incorporate additional information, e.g. on the architectural constraints. In the current implementation, the RTS is unable to switch between the baseline and the SC mechanism at run time.

In non-strict languages, operational behaviour is only loosely coupled with the static code-level denotational semantics. This makes optimisation more challenging in this setting. In this work we have not utilised any static analysis which could potentially give a clue to the granularity associated with sparks. We also do not use cost models or architectural information in our heuristics. The literature suggests that such features can be beneficially incorporated into some of the decisions.

7.3 Future Work

Given the above limitations, we identify serveral directions for future research.

Operationally Deterministic Work Stealing We would like to explore ways to implement operationally deterministic work stealing, for instance by using overlapping multicast groups or MPI comm-groups, which would enable better predictability and cost analysis by splitting the set of all PEs into subgroups and allowing stealing only in these subgroups using fixed communication paths. Because the subgroups overlap every PE would be reachable from every other, even though the communication would be indirect. The groups could share more knowledge amongst its members to improve performance. This would also enable more flexible study of adaptation to NUMA architectures, as the groups could be defined to match NUMA regions. Additionally, a gossip or a publish/subscribe overlay could be investigated for dissemination of load and other relevant system-level information as well as architectural information within the group and among the groups. This can be viewed as orthogonal to the GUM protocol and treating it as such would result in a more modular implementation.

Investigating Accuracy-Coverage Trade-Off for History-Based Stealing Further exploration is needed to evaluate the accuracy-coverage trade-off and invalidation interval selection on a broader range of architectures using additional larger applications. Coverage can be measured as the fraction of the PEs of total, for which the information is up-to-date. This can be either added to the global profiling as a running average updated every time the information is used or as part of the census-based profiling providing a more detailed view over time. Accuracy is more difficult to assess, as it requires a golden standard to compare against. This could potentially be achieved if execution replay work [83] can be ported to GUM. However, using an operationally non-deterministic work stealing variant may complicate this, making the work from previous paragraph well-suited to complement this direction.

Automating Spark Colocation Following up on Spark Colocation, introducing the encoding, which was manually done for this work, could be automated based on existing techniques and would entail replacing the `par` annotations with `parEnc` whilst keeping track of parallelism sources as a pass in the compiler or using a preprocessor. This would facilitate further study and the use of the approach. Moreover, it would be of interest to investigate the effects on other kinds of parallel architectures, using larger parallel applications with different parallelism patterns, and to compare different matching functions ways to encode ancestry. In particular it appears useful to be able to flexibility switch between the baseline and SC at runtime based on the number of PEs and system-level parameters such as spark pool size.

Graph Reduction in Hardware Observing recent trends in continuing FPGA clock and memory size scaling, it seems worthwhile to revisit past work on graph reduction hardware using this opportunity. Recent work on Reduceron [185] and PilGRIM [39] appears promising. We believe that graph reduction is particularly suited for fine-grained parallel execution offered by FPGA-based approach, which has the benefit of much faster prototyping life cycle than the graph reduction machine projects had in the past. Defining a parallel machine that is able to use multiple Reduceron cores could be the first step in this direction.

Bibliography

[1] S. Abramsky and R. Sykes. SECD-M: A virtual machine for applicative programming. In *Conference on Functional Programming Languages and Computer Architecture*, pages 81–98. Springer, 1985.

[2] G. A. Agha. Actors: A model of concurrent computation in distributed systems. Technical report, Massachusetts Institute of Technology, Cambridge, Artificial Intelligence Lab, 1985.

[3] M. Aljabri, H.-W. Loidl, and P. Trinder. The design and implementation of GUMSMP: a multilevel parallel Haskell implementation. In *Proc. of 25th ACM Symp. on Implementation & Application of Functional Languages*, pages 37–48, 2013.

[4] M. Aljabri, H.-W. Loidl, and P. Trinder. Distributed vs. shared heap, parallel Haskell implementations on shared memory machines. In *Proc. of Symp. on Trends in Functional Programming*, Univ. of Utrecht, The Netherlands, 2014.

[5] T. E. Anderson, D. E. Culler, and D. Patterson. A case for NOW (networks of workstations). *IEEE micro*, 15(1):54–64, 1995.

[6] B. Archibald, P. Maier, R. Stewart, P. Trinder, and J. De Beule. Towards generic scalable parallel combinatorial search. In *Proceedings of the International Workshop on Parallel Symbolic Computation*, page 6. ACM, 2017.

[7] G. Argo. Improving the three instruction machine. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 100–115. ACM, 1989.

[8] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[9] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, 2005.

[10] Arvind and R. S. Nikhil. Id: A language with implicit parallelism. In *A Comparative Study of Parallel Programming Languages*, pages 169–215. Elsevier, 1992.

[11] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, et al. The landscape of parallel computing research: A view from Berkeley. Technical report, UCB/EECS-2006-183, EECS Dept, University of California, Berkeley, 2006.

[12] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *CACM*, 52:56–67, October 2009.

[13] M. Aswad, P. Trinder, and H.-W. Loidl. Architecture aware parallel programming in Glasgow Parallel Haskell (GPH). *Procedia Computer Science*, 9:1807–1816, 2012.

[14] L. Augustsson and T. Johnsson. Parallel graph reduction with the (v, G)-machine. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 202–213. ACM, 1989.

[15] L. Augustsson and T. Johnsson. The Chalmers Lazy-ML compiler. *The Computer Journal*, 32(2):127–141, 1989.

[16] J. Backus. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs. *CACM*, 21(8):613–641, 1978.

[17] H. C. Baker Jr and C. Hewitt. The incremental garbage collection of processes. In *ACM SIGPLAN Notices*, volume 12, pages 55–59, 1977.

[18] H. Barendregt, W. Dekkers, and R. Statman. *Lambda calculus with types*. Cambridge University Press, 2013.

[19] H. P. Barendregt. *The lambda calculus*, volume 3. North-Holland Amsterdam, 1984.

[20] P. S. Barth, R. S. Nikhil, et al. M-structures: extending a parallel, non-strict, functional language with state. In *Conference on Functional Programming Languages and Computer Architecture*, pages 538–568. Springer, 1991.

[21] E. Belikov. History-based adaptive work distribution. In *Proc. of Imperial College Computing Student Workshop*, volume 43 of *OpenAccess Series in Informatics (OASIcs)*, pages 3–10. Leibniz-Zentrum fuer Informatik, 2014.

[22] E. Belikov. Hitchhiker's guide to GUM hacking. Technical Report HW-MACS-TR-0112, Dept of Computer Science, Heriot-Watt University, Dec. 2015.

[23] E. Belikov. Language Run-time Systems: an Overview. In *Proc. of Imperial College Computing Student Workshop*, volume 49 of *OpenAccess Series in Informatics (OASIcs)*, pages 3–12. Leibniz-Zentrum fuer Informatik, 2015.

[24] E. Belikov, P. Deligiannis, P. Totoo, M. Aljabri, and H.-W. Loidl. A survey of high-level parallel programming models. Technical Report HW-MACS-TR-0103, Dept of Computer Science, Heriot-Watt University, Dec. 2013.

[25] E. Belikov, H.-W. Loidl, and G. Michaelson. Towards a characterisation of parallel functional applications. In *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering, Dresden, Germany*, pages 146–153, 2015.

[26] E. Belikov, H.-W. Loidl, and G. Michaelson. Colocation of potential parallelism in a distributed adaptive run-time system for parallel Haskell. In *Proceedings of the International Symposium on Trends in Functional Programming, Gothenburg, Sweden*, pages 1–19. Springer, 2018.

[27] J. Berthold, A. Al Zain, and H.-W. Loidl. Scheduling Light-Weight Parallelism in ArTCoP. In P. Hudak and D. Warren, editors, *Practical Aspects of Declarative Languages*, volume 4902 of *Lecture Notes in Computer Science*, pages 214–229. Springer Berlin / Heidelberg, 2008.

[28] J. Berthold, M. Dieterle, O. Lobachev, and R. Loogen. Distributed memory programming on many-cores a case study using Eden Divide-&-Conquer skeletons. In *22nd Intl Conf. on Architecture of Computing Systems (ARCS)*, pages 1–9. VDE, 2009.

[29] J. Berthold, M. Dieterle, and R. Loogen. Implementing parallel Google map-reduce in Eden. In *European Conference on Parallel Processing*, pages 990–1002. Springer, 2009.

[30] J. Berthold, H.-W. Loidl, and K. Hammond. PAEAN: Portable runtime support for physically-shared-nothing architectures in parallel Haskell dialects. *Journal of Functional Programming*, 2016.

[31] D. Bevan. An efficient reference counting solution to the distributed garbage collection problem. *Parallel Computing*, 9(2):179–192, 1989.

[32] C. Bienia, S. Kumar, and K. Li. PARSEC vs. SPLASH-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *Proc. of the IEEE Intl Symposium on Workload Characterization*, pages 47–56, 2008.

[33] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: hardware design in Haskell. In *ACM SIGPLAN Notices*, volume 34, pages 174–184. ACM, 1998.

[34] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.

[35] J. Blazewicz, K. H. Ecker, G. Schmidt, and J. Weglarz. *Scheduling in computer and manufacturing systems*. Springer Science & Business Media, 2012.

[36] G. E. Blelloch. Nesl: A nested data-parallel language (version 3.1). Technical report, Carnegie-Mellon University Pittsburgh PA School of Computer Science, 1995.

[37] A. Bloss, P. Hudak, and J. Young. Code optimizations for lazy evaluation. *Lisp and Symbolic Computation*, 1(2):147–164, 1988.

[38] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming (PPoPP'95)*, pages 207–216, 1995.

[39] A. Boeijink, P. K. F. Hölzenspies, and J. Kuper. Introducing the PilGRIM: A processor for executing lazy functional languages. In *22nd International Symposium on the Implementation and Application of Functional Languages, Alphen aan den Rijn, The Netherlands, September 1-3, 2010, Revised Selected Papers*, pages 54–71. Springer, 2010.

[40] S. Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, pages 746–749. ACM, 2007.

[41] S. Breitinger, U. Klusik, R. Loogen, Y. Ortega-Mallén, and R. Pena. DREAM: the distributed Eden abstract machine. In *Symposium on Implementation and Application of Functional Languages*, pages 250–269. Springer, 1997.

[42] P. Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–241, 1970.

[43] A. Brodtkorb, C. Dyken, T. Hagen, J. Hjelmervik, and O. Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1–33, May 2010.

[44] J. M. Bull. A hierarchical classification of overheads in parallel programs. In *Software Engineering for Parallel and Distributed Systems*, pages 208–219. Springer, 1996.

[45] W. H. Burge. Recursive programming techniques. 1975.

[46] G. L. Burn. Implementing lazy functional languages on parallel architectures. *Parallel Computers— Object-Oriented, Functional, Logic, Series in Parallel Computing*, pages 101–140, 1990.

[47] G. L. Burn, S. Peyton Jones, and J. D. Robson. The Spineless G-machine. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 244–258. ACM, 1988.

[48] R. M. Burstall, D. B. MacQueen, and D. T. Sannella. Hope: An experimental applicative language. In *Proceedings of the 1980 ACM conference on LISP and functional programming*, pages 136–143. ACM, 1980.

[49] F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, pages 187–194. ACM, 1981.

[50] J. M. Calderon Trilla. *Improving Implicit Parallelism*. PhD thesis, University of York, 2015.

[51] D. Cann and J. Feo. SISAL versus FORTRAN: A comparison using the Livermore Loops. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 626–636. IEEE Computer Society Press, 1990.

[52] L. Cardelli. *The functional abstract machine*. AT&T Bell Laboratories. Computing Science, 1984.

[53] G. D. Carlow. Architecture of the space shuttle primary avionics software system. *Communications of the ACM*, 27(9):926–936, 1984.

[54] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Acm Sigplan Notices*, volume 40, pages 519–538. ACM, 2005.

[55] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *Proc. of the 17th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 21–28, 2005.

[56] G. M.-B. Chaslot, M. H. Winands, and H. J. van Den Herik. Parallel Monte-Carlo tree search. In *International Conference on Computers and Games*, pages 60–71. Springer, 2008.

[57] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell Broadband Engine architecture and its first implementation – a performance view. *IBM Journal of Research and Development*, 51(5):559–572, 2007.

[58] D. Y. Cheng. A survey of parallel programming languages and tools. *NASA Ames Research Center*, 1993.

[59] G. Chrysos. Intel Xeon Phi coprocessor - the architecture. *Intel Whitepaper*, 176, 2014.

[60] F. Chung, R. Graham, R. Bhagwan, S. Savage, and G. M. Voelker. Maximizing data locality in distributed systems. *Journal of Computer and System Sciences*, 72(8):1309–1316, 2006.

[61] A. Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, pages 346–366, 1932.

[62] A. Church. The calculi of lambda-conversion, volume 6 of Annals of Mathematics Studies, 1941.

[63] A. Church and J. B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936.

[64] C. Clack and S. Peyton Jones. Strictness analysis – a practical approach. In *Conference on Functional Programming Languages and Computer Architecture*, pages 35–49. Springer, 1985.

[65] C. Clack and S. Peyton Jones. The four-stroke reduction engine. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 220–232, 1986.

[66] M. Cole. *Algorithmic Skeletons: Structural Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. MIT Press, 1989.

[67] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen. Solving large, irregular graph problems using adaptive work-stealing. In *Parallel Processing, 2008. ICPP'08. 37th International Conference on*, pages 536–545. IEEE, 2008.

[68] P. Costa, H. Ballani, and D. Narayanan. Rethinking the network stack for rack-scale computers. In *HotCloud*, 2014.

[69] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of computer programming*, 8(2):173–202, 1987.

[70] M. Cripps, J. Darlington, A. Field, P. Harrison, and M. Reeve. The design and implementation of ALICE: A parallel graph reduction machine. In *Proceedings of the Workshop on Graph Reduction*, 1987.

[71] H. B. Curry, R. Feys, W. Craig, J. R. Hindley, and J. P. Seldin. *Combinatory logic*, volume 1. North-Holland Amsterdam, 1958.

[72] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[73] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM, 1987.

[74] J. Diaz, C. Munoz-Caro, and A. Nino. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Transactions on parallel and distributed systems*, 23(8):1369–1386, 2012.

[75] S. Diehl, P. Hartel, and P. Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16(7):739–751, 2000.

[76] J. J. Dongarra, H. W. Meuer, E. Strohmaier, et al. Top500 supercomputer sites. *Supercomputer*, 13:89–111, 1997. www.top500.org (last accessed: 28.06.2018).

[77] A. R. Du Bois, H.-W. Loidl, and P. Trinder. Thread migration in a parallel graph reducer. In *Implementation of Functional Languages*, pages 199–214. Springer, 2002.

[78] R. Duncan. A survey of parallel computer architectures. *Computer*, 23(2):5–16, 1990.

[79] D. L. Eager, E. D. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance evaluation*, 6(1):53–68, 1986.

[80] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 365–376. ACM, 2011.

[81] J. Fairbairn and S. Wray. TIM: A simple, lazy abstract machine to execute supercombinators. In *Conference on Functional Programming Languages and Computer Architecture*, pages 34–45. Springer, 1987.

[82] J. T. Feo, D. C. Cann, and R. R. Oldehoeft. A report on the SISAL language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, 1990.

[83] H. Ferreiro, V. Janjic, L. M. Castro, and K. Hammond. Repeating history: execution replay for parallel Haskell programs. In *International Symposium on Trends in Functional Programming*, pages 231–246. Springer, 2012.

[84] A. Field and P. Harrison. *Functional programming*. Addison-Wesley, 1988.

[85] M. Fluet, M. Rainey, and J. Reppy. A scheduling framework for general-purpose parallel languages. In *ACM SIGPLAN Notices*, volume 43, pages 241–252, 2008.

[86] M. Fluet, M. Rainey, J. Reppy, A. Shaw, and Y. Xiao. Manticore: A heterogeneous parallel language. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 37–44. ACM, 2007.

[87] M. J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[88] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *Computers, IEEE Transactions on*, C-21(9):948–960, sept. 1972.

[89] M. J. Flynn and K. W. Rudd. Parallel architectures. *ACM Comput. Surv.*, 28(1):67–70, Mar. 1996.

[90] I. Foster. *Designing and building parallel programs*, volume 78. Addison Wesley Publishing Company Boston, 1995.

[91] I. Foster and C. Kesselman. *The Grid 2: Blueprint for a new computing infrastructure*. Elsevier, 2003.

[92] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. The physiology of the grid. *Grid computing: making the global infrastructure a reality*, pages 217–249, 2003.

[93] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *International journal of high performance computing applications*, 15(3):200–222, 2001.

[94] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud Computing and Grid Computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE'08*, pages 1–10. Ieee, 2008.

[95] V. W. Freeh, D. K. Lowenthal, and G. R. Andrews. Distributed Filaments: Efficient fine-grain parallelism on a cluster of workstations. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 15. USENIX Association, 1994.

[96] D. P. Friedman and D. S. Wise. *CONS should not evaluate its arguments.* Computer Science Department, Indiana University, 1976.

[97] D. P. Friedman and D. S. Wise. *The impact of applicative programming on multiprocessing.* Indiana University, Computer Science Department, 1976.

[98] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine: A User's Guide and Tutorial for networked Parallel Computing.* MIT Press, 1994.

[99] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):96, 1992.

[100] B. Goldberg. Multiprocessor execution of functional programs. *International Journal of Parallel Programming*, 17(5):425–473, 1988.

[101] B. Goldberg and P. Hudak. Alfalfa: distributed graph reduction on a hypercube multiprocessor. In *Graph Reduction*, pages 94–113. Springer, 1987.

[102] S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, 1996.

[103] H. Gonzalez-Velez and M. Leyton. A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers. *Software: Practice and Experience*, 40(12), 2010.

[104] C. Gregg and K. Hazelwood. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 134–144. IEEE, 2011.

[105] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[106] L. Gwennap. Adapteva: More flops, less watts. *Microprocessor Report*, 6(13):11–02, 2011.

[107] R. H. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[108] K. Hammond. Parallel functional programming: An introduction. In *Proc. PASCO*, volume 94, pages 181–193. World Scientific, 1994.

[109] K. Hammond. Why parallel functional programming matters: Panel statement. In *Reliable Software Technologies Ada-Europe*, pages 201–205. Springer, 2011.

[110] K. Hammond, H.-W. Loidl, and P. Trinder. Parallel Cost Centre Profiling. In *Proceedings of the Glasgow Workshop on Functional Programming*, Ullapool, Scotland, Sept. 1997.

[111] K. Hammond and G. Michaelson. *Research directions in parallel functional programming*. Springer Science & Business Media, 2012.

[112] T. Harris. Hardware trends: Challenges and opportunities in distributed computing. *ACM SIGACT News*, 46(2):89–95, 2015.

[113] T. Harris, S. Marlow, and S. P. Jones. Haskell on a Shared-Memory Multiprocessor. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, Haskell '05, pages 49–61, New York, NY, USA, 2005. ACM.

[114] T. Harris and S. Singh. Feedback directed implicit parallelism. In *ACM SIGPLAN Notices*, volume 42, pages 251–264. ACM, 2007.

[115] P. Henderson. *Functional programming: application and implementation*. Prentice-Hall, 1980.

[116] P. Henderson and J. H. Morris Jr. A lazy evaluator. In *Proceedings of the 3rd ACM SIGACT-SIGPLAN symposium on Principles on programming languages*, pages 95–103. ACM, 1976.

[117] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[118] J. L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[119] C. Hewitt, P. Bishop, and R. Steiger. Session 8 Formalisms for Artificial Intelligence – a universal modular actor formalism for artificial intelligence. In *Advance Papers of the Conference*, volume 3, page 235. Stanford Research Institute, 1973.

[120] R. Hindley. The principle type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

[121] Z. Hu, J. Hughes, and M. Wang. How functional programming mattered. *National Science Review*, 2(3):349–370, 2015.

[122] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys (CSUR)*, 21(3):359–411, 1989.

[123] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A History of Haskell: Being Lazy With Class. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 12–1–12–55. ACM, 2007.

[124] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, et al. Report on the pro-

gramming language Haskell: a non-strict, purely functional language version 1.2. *ACM SIGPLAN Notices*, 27(5):1–164, 1992.

[125] P. Hudak and L. Smith. Para-functional programming: a paradigm for programming multiprocessor systems. In *Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 243–254. ACM, 1986.

[126] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[127] R. J. M. Hughes. Super-combinators a new implementation method for applicative languages. In *Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 1–10. ACM, 1982.

[128] R. J. M. Hughes. *The design and implementation of programming languages*. PhD thesis, University of Oxford, 1983.

[129] G. Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, 1999.

[130] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, Inc., 1991.

[131] V. Janjic and K. Hammond. Granularity-aware work-stealing for computationally-uniform grids. In *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*, pages 123–134. IEEE, 2010.

[132] B. Jeff. Big.LITTLE system architecture from ARM: saving power through heterogeneous multiprocessing and task context migration. In *Proceedings of the 49th Annual Design Automation Conference*, pages 1143–1146. ACM, 2012.

[133] J. Jeffers and J. Reinders. *Intel Xeon Phi coprocessor high performance programming*. Newnes, 2013.

[134] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Conference on Functional programming languages and computer architecture*, pages 190–203. Springer, 1985.

[135] R. Jones, A. Hosking, and E. Moss. *The garbage collection handbook: the art of automatic memory management*. Chapman & Hall/CRC, 2012.

[136] P. H. Kelly. *Functional programming for loosely-coupled multiprocessors*. MIT Press, 1989.

[137] H. Kingdon, D. R. Lester, and G. L. Burn. The HDG-machine: a highly distributed graph-reducer for a transputer network. *The Computer Journal*, 34(4):290–300, 1991.

[138] S. C. Kleene et al. λ-definability and recursiveness. *Duke Mathematical Journal*, 2(2):340–353, 1936.

[139] W. Kluge. *Abstract Computing Machines: A Lambda Calculus Perspective*. Springer Science & Business Media, 2006.

[140] D. A. Kranz, R. H. Halstead Jr, and E. Mohr. Mul-T: A high-performance parallel Lisp. In *ACM SIGPLAN Notices*, volume 24, pages 81–90, 1989.

[141] J.-L. Krivine. A call-by-name lambda-calculus machine. *Higher-order and symbolic computation*, 20(3):199–207, 2007.

[142] H.-T. Kung. Why systolic architectures? *IEEE computer*, 15(1):37–46, 1982.

[143] I. Kuon, R. Tessier, and J. Rose. FPGA architecture: Survey and challenges. *Foundations and Trends in Electronic Design Automation*, 2(2):135–253, 2008.

[144] C. Lameter. An overview of non-uniform memory access. *Communications of the ACM*, 56(9):59–54, 2013.

[145] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1964.

[146] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[147] P. J. Landin. A generalization of jumps and labels. *Higher-Order and Symbolic Computation*, 11(2):125–143, 1998.

[148] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, pages 75–86. IEEE, 2004.

[149] E. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, May 2006.

[150] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund, et al. Debunking the 100x GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. *ACM SIGARCH Computer Architecture News*, 38(3):451–460, 2010.

[151] B. Lepers, V. Quéma, and A. Fedorova. Thread and memory placement on {NUMA} systems: Asymmetry matters. In *2015 {USENIX} Annual Technical Conference ({USENIX}{ATC} 15)*, pages 277–289, 2015.

[152] X. Leroy. *The ZINC experiment: an economical implementation of the ML language*. PhD thesis, INRIA, 1990.

[153] X. Leroy. The Caml Light system release 0.74. *URL: http://caml.inria.fr*, 1997.

[154] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The objective caml system release 3.11. *Documentation and users manual. INRIA*, 2008.

[155] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java virtual machine specification*. Pearson Education, 2014.

[156] B. Liskov and L. Shrira. *Promises: linguistic support for efficient asynchronous procedure calls in distributed systems*, volume 23. ACM, 1988.

[157] H.-W. Loidl. *Granularity in Large-Scale Parallel Functional Programming.* PhD thesis, Department of Computing Science, University of Glasgow, Mar. 1998.

[158] H.-W. Loidl. The virtual shared memory performance of a parallel graph reducer. In *2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 311–311, 2002.

[159] H.-W. Loidl and K. Hammond. On the granularity of divide-and-conquer parallelism. In *Functional Programming*, page 8, 1995.

[160] H.-W. Loidl and K. Hammond. Making a packet: cost-effective communication for a parallel graph reducer. In *Symposium on Implementation and Application of Functional Languages*, pages 184–199. Springer, 1996.

[161] H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Peña, S. Priebe, et al. Comparing parallel functional languages: Programming and performance. *Higher-Order and Symbolic Computation*, 16(3):203–251, 2003.

[162] H.-W. Loidl, P. Trinder, and C. Butz. Tuning task granularity and data locality of data parallel GpH programs. *Parallel Processing Letters*, 11(04):471–486, 2001.

[163] H.-W. Loidl and P. W. Trinder. Engineering Large Parallel Functional Programs. In *Implementation of Functional Languages, 1997*, LNCS. Springer-Verlag, Sept. 1997.

[164] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel Functional Programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[165] I. L. Markov. Limits on fundamental limits to computation. *Nature*, 512(7513):147–154, 2014.

[166] S. Marlow. *Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming.* O'Reilly, 2013.

[167] S. Marlow, P. Maier, H.-W. Loidl, M. Aswad, and P. Trinder. Seq no more: better strategies for parallel Haskell. In *Proc. of the 3rd ACM Symposium on Haskell*, pages 91–102, 2010.

[168] S. Marlow, R. Newton, and S. Peyton Jones. A Monad for Deterministic Parallelism. In *Haskell '11, Tokyo, Japan*, pages 71–82. ACM Press, 2011.

[169] S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore Haskell. In *ACM SIGPLAN Notices*, volume 44, pages 65–78, 2009.

[170] S. Marlow and S. L. Peyton Jones. The Glasgow Haskell Compiler. *The Architecture of Open Source Applications*, 2, 2012.

[171] S. Marlow, A. R. Yakushev, and S. Peyton Jones. Faster laziness using dynamic pointer tagging. In *ACM SIGPLAN Notices*, volume 42, pages 277–288. ACM, 2007.

[172] S. Marlow (Ed.). Haskell 2010 language report. 2010. `http://www.haskell.org/onlinereport/haskell2010`.

[173] D. L. McBurney and M. R. Sleep. Transputer-based experiments with the ZAPP architecture. In *International Conference on Parallel Architectures and Languages Europe*, pages 242–259. Springer, 1987.

[174] D. L. McBurney and M. R. Sleep. Experiments with a virtual tree machine using transputers. In *System Sciences, 1989. Vol. I: Architecture Track, Proceedings of the Twenty-Second Annual Hawaii International Conference on*, volume 1, pages 355–364. IEEE, 1989.

[175] J. D. McCalpin et al. Memory bandwidth and machine balance in current high performance computers. *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, 1995:19–25, 1995.

[176] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[177] J. McCarthy. History of LISP. In *History of programming languages I*, pages 173–185. ACM, 1978.

[178] J. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, I. Dobes, and P. Hohensee. SISAL: streams and iteration in a single-assignment language. language reference manual, version 1. 1. Technical report, Lawrence Livermore National Lab., CA (USA), 1983.

[179] E. Meijer and J. Gough. Technical overview of the common language runtime. *language*, 29:7, 2001.

[180] R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[181] R. Milner. *The definition of standard ML: revised*. MIT press, 1997.

[182] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[183] E. Mohr, D. Kranz, R. Halstead Jr, et al. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, 1991.

[184] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 8:144–117, 1965.

[185] M. Naylor and C. Runciman. The Reduceron reconfigured and re-evaluated. *Journal of Functional Programming*, 22(4-5):574–613, 2012.

[186] B. Nichols, D. Buttlar, and J. Farrell. *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.", 1996.

[187] R. S. Nikhil, K. K. Pingali, et al. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4):598–632, 1989.

[188] E. Nöcker, J. Smetsers, M. C. van Eekelen, and M. J. Plasmeijer. Concurrent Clean. In *International Conference on Parallel Architectures and Languages Europe*, pages 202–219. Springer, 1991.

[189] J. Ousterhout. Why threads are a bad idea (for most purposes). In *Presentation given at the 1996 Usenix Annual Technical Conference*, volume 5. San Diego, CA, USA, 1996.

[190] J. Owens, D. Luebke, N. Govindaraju, et al. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[191] P. Pacheco. *An introduction to parallel programming*. Elsevier, 2011.

[192] W. Partain. The nofib benchmark suite of Haskell programs. In *Functional Programming, Glasgow 1992*, pages 195–202. Springer, 1993.

[193] S. Peyton Jones. *The implementation of functional programming languages (prentice-hall international series in computer science)*. Prentice-Hall, Inc., 1987.

[194] S. Peyton Jones. Parallel implementations of functional programming languages. *The Computer Journal*, 32(2):175–186, 1989.

[195] S. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(02):127–202, 1992.

[196] S. Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[197] S. Peyton Jones. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. 2008.

[198] S. Peyton Jones, C. Clack, and J. Salkild. High-performance parallel graph reduction. In *PARLE'89 Parallel Architectures and Languages Europe*, pages 193–206. Springer, 1989.

[199] S. Peyton Jones, C. Clack, J. Salkild, and M. Hardie. GRIP – a high-performance architecture for parallel graph reduction. In *Functional Programming Languages and Computer Architecture*, pages 98–112. Springer, 1987.

[200] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *POPL*, volume 96, pages 295–308, 1996.

[201] S. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell Compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, volume 93, 1993.

[202] S. Peyton Jones, N. Ramsey, and F. Reig. C--: A portable assembly language that supports garbage collection. In *Principles and Practice of Declarative Programming*, pages 1–28. Springer, 1999.

[203] D. Plainfossé and M. Shapiro. A survey of distributed garbage collection techniques. In *Memory Management*, pages 211–249. Springer, 1995.

[204] G. D. Plotkin. A structural approach to operational semantics. 1981.

[205] M. Quinn. *Parallel Programming using C with MPI and OpenMP*. McGraw-Hill, 2003.

[206] F. Rabhi and S. Gorlatch. *Patterns and skeletons for parallel and distributed computing*. Springer, 2003.

[207] C. Ramey. Tile-GX100 manycore processor: Acceleration interfaces and architecture. In *Hot Chips 23rd Symposium*, pages 1–21. IEEE, 2011.

[208] D. A. Reed and J. Dongarra. Exascale computing and big data. *Communications of the ACM*, 58(7):56–68, 2015.

[209] D. Ridge, D. Becker, P. Merkey, and T. Sterling. Beowulf: harnessing the power of parallelism in a pile-of-PCs. In *Aerospace Conference, 1997. Proceedings., IEEE*, volume 2, pages 79–91. IEEE, 1997.

[210] J. Ross and A. E. Phelps. Computing convolutions using a neural network processor, July 4 2017. US Patent 9,697,463.

[211] D. Rushall. *Task exposure in the parallel implementation of functional programming Languages*. PhD thesis, University of Manchester, 1995.

[212] M. Scheevel. NORMA: a graph reduction processor. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 212–219. ACM, 1986.

[213] S.-B. Scholz. Single-assignment C – functional programming using imperative style. In J. Glauert, editor, *6th International Workshop on Implementation of Functional Languages (IFL'94)*, pages 211–2113. University of East Anglia, Norwich, England, UK, 1994.

[214] S.-B. Scholz. Single Assignment C: efficient support for high-level array operations in a functional setting. *Journal of Functional Programming*, 13(6):1005–1059, 2003.

[215] W. Schreiner. Parallel functional programming – an annotated bibliography. 1993.

[216] D. S. Scott and C. Strachey. *Toward a mathematical semantics for computer languages*, volume 1. Oxford University Computing Laboratory, Programming Research Group, 1971.

[217] H. Seidl and R. Wilhelm. Probabilistic load balancing for parallel graph reduction. In *TENCON'89. Fourth IEEE Region 10 International Conference*, pages 879–884. IEEE, 1989.

[218] P. Sestoft. *Analysis and Efficient Implementation of Functional Languages*. PhD thesis, DIKU, University of Copenhagen, 1991.

[219] E. Shapiro. The fifth generation project – a trip report. *Communications of the ACM*, 26(9):637–641, 1983.

[220] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys (CSUR)*, 21(3):413–510, 1989.

[221] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating system concepts essentials*. John Wiley & Sons, Inc., 2014.

[222] D. Skillicorn. A taxonomy for computer architectures. *Computer*, 21(11):46–57, nov. 1988.

[223] D. B. Skillicorn and D. Talia. Models and Languages for Parallel Computation. *ACM Computing Surveys*, 30(2):123–169, June 1998.

[224] M. Sloman. Policy driven management for distributed systems. *Journal of network and Systems Management*, 2(4):333–360, 1994.

[225] D. J. Sorin, M. D. Hill, and D. A. Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3):1–212, 2011.

[226] M. Steuwer, T. Remmelg, and C. Dubach. Lift: a functional data-parallel IR for high-performance GPU code generation. In *2017 IEEE/ACM Intl Symposium on Code Generation and Optimization (CGO)*, pages 74–85. IEEE, 2017.

[227] J. E. Stoy. *Denotational semantics: the Scott-Strachey approach to programming language theory*. MIT press, 1977.

[228] H. Sundell and P. Tsigas. Lock-free deques and doubly linked lists. *Journal of Parallel and Distributed Computing*, 68(7):1008–1020, 2008.

[229] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs journal*, 30(3):202–210, 2005.

[230] D. Terei and M. Chakravarty. An LLVM backend for GHC. In *ACM SIGPLAN Notices*, volume 45, pages 109–120, 2010.

[231] P. Totoo. *Parallel evaluation strategies for lazy data structures in Haskell*. PhD thesis, Heriot-Watt University, 2016.

[232] P. Totoo, P. Deligiannis, and H.-W. Loidl. Haskell vs. F# vs. Scala: a high-level language features and parallelism support comparison. In *Proceedings of the 1st ACM SIGPLAN workshop on Functional High-Performance Computing*, pages 49–60. ACM, 2012.

[233] K. R. Traub. *Implementation of non-strict functional programming languages*. MIT Press, 1991.

[234] P. C. Treleaven. *Parallel Computers: Object-Oriented, Functional, Logic*. 1990.

[235] P. Trinder, E. Barry Jr, M. Davis, K. Hammond, S. Junaidu, U. Klusik, H.-W. Loidl, and S. Peyton Jones. GpH: An architecture-independent functional language. *IEEE Transactions on Software Engineering*, 1998.

[236] P. Trinder, K. Hammond, H.-W. Loidl, and S. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, January 1998.

[237] P. Trinder, K. Hammond, J. Mattson Jr, A. Partridge, and S. Peyton Jones. GUM: a portable parallel implementation of Haskell. In *Proc. of PLDI*, pages 79–88, 1996.

[238] P. Trinder, H.-W. Loidl, E. Barry Jr, K. Hammond, U. Klusik, S. Peyton Jones, and A. Rebon Portillo. The multi-architecture performance of the parallel functional language GpH. In *Proc. of Euro-Par*, volume 1900 of *LNCS*, pages 739–743, 2000.

[239] P. W. Trinder, H.-W. Loidl, and R. F. Pointon. Parallel and distributed Haskells. *Journal of Functional Programming*, 12(4-5):469–510, 2002.

[240] E. R. Tufte and P. Graves-Morris. *The visual display of quantitative information*, volume 2. Graphics Press Cheshire, CT, 1983.

[241] A. M. Turing. Computability and λ-definability. *The Journal of Symbolic Logic*, 2(4):153–163, 1937.

[242] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(1):230–265, 1937.

[243] D. A. Turner. A new implementation technique for applicative languages. *Software: Practice and Experience*, 9(1):31–49, 1979.

[244] D. A. Turner. The semantic elegance of applicative languages. In *Proceedings of the 1981 conference on Functional Programming Languages and Computer Architecture*, pages 85–92. ACM, 1981.

[245] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Conference on Functional Programming Languages and Computer Architecture*, pages 1–16. Springer, 1985.

[246] D. A. Turner. Some history of functional programming languages. In *International Symposium on Trends in Functional Programming*, pages 1–20. Springer, 2012.

[247] S. Ulam. John von Neumann 1903-1957. *Bulletin of the American mathematical society*, 64(3):1–49, 1958.

[248] P. Van Roy et al. Programming paradigms for dummies: What every programmer should know. *New computational paradigms for computer music*, 104, 2009.

[249] P. Van Roy, S. Haridi, P. Brand, G. Smolka, M. Mehl, and R. Scheidhauer. Mobile objects in distributed Oz. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(5):804–851, 1997.

[250] S. R. Vegdahl. A survey of proposed architectures for the execution of functional languages. *Computers, IEEE Transactions on*, 100(12):1050–1071, 1984.

[251] J. Vuillemin. Correct and optimal implementations of recursion in a simple programming language. *Journal of Computer and System Sciences*, 9(3):332–354, 1974.

[252] P. Wadler. The essence of functional programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14. ACM, 1992.

[253] C. P. Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus.* PhD thesis, University of Oxford, 1971.

[254] M. D. Wael, S. Marr, B. D. Fraine, T. V. Cutsem, and W. D. Meuter. Partitioned global address space languages. *ACM Computing Surveys (CSUR)*, 47(4):62, 2015.

[255] K. B. Wheeler, R. C. Murphy, and D. Thain. Qthreads: An API for programming with millions of lightweight threads. In *Proceedings of International Symposium on Parallel and Distributed Processing*, pages 1–8. IEEE, 2008.

[256] D. Williamson, R. Parker, and J. Kendrick. The box plot: a simple visual method to interpret data. *Annals of internal medicine*, 110(11):916–921, 1989.

[257] W. Wolf, A. A. Jerraya, and G. Martin. Multiprocessor system-on-chip (MPSoC) technology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1701–1713, 2008.

[258] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[259] Y. Xie and G. Loh. Dynamic classification of program memory behaviors in CMPs. In *the 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2008.

[260] J. Yang and Q. He. Scheduling parallel computations by work stealing: A survey. *International Journal of Parallel Programming*, 46(2):173–197, 2018.

[261] A. Yonezawa and M. Tokoro. Object-oriented concurrent programming. 1986.

Appendix A

Applications and Measurements

This appendix section presents selected data and information how to obtain the source code and the data.

A.1 Source Code

Application and RTS source code as well as the data sets are available as part of the *e-thesis*. Additionally, the application source code is available on GitHub: `https://github.com/jevbelikov/gum-apps`.

A.2 Message Counts

The tables below present message counts from a selected run with execution time closest to the calculated median. All history-based data is for run with an invalidation interval of 1000 ms, unless otherwise specified. For summary Figures and discussion refer to Chapter 5.

A.2.1 History-Based Stealing

Table A.1: Summary of Sent Messages for `parfib`

number of PEs	baseline total			history-based (`-qz100`) total		
		`FISH`	`SCHEDULE`		`FISH`	`SCHEDULE`
64	6719	3573	784	7637	4785	712
80	8254	4754	872	9527	5474	1010
96	11547	6857	1171	10034	6387	910
112	12830	8143	1166	13782	7936	1459
128	13298	7546	1437	13464	7829	1408
144	15062	8577	1620	13916	7802	1527
160	14000	6922	1768	15688	9653	1507
176	17671	11116	1637	16635	9200	1855
192	18459	10649	1950	18968	11212	1935
208	21214	13222	1995	19840	12208	1904
224	21145	12942	2048	22597	13696	2223
240	26910	17935	2240	24949	15840	2271
256	27348	17918	2352	25288	16978	2074

Table A.2: Summary of Sent Messages for `coins`

number of PEs	baseline total			history-based total		
		`FISH`	`SCHEDULE`		`FISH`	`SCHEDULE`
64	62318	24597	9426	58621	26127	8120
80	53031	23166	7462	82806	37351	11360
96	82785	31024	12935	67446	28373	9745
112	79698	27777	12976	86520	38917	11898
128	76663	25144	12875	110678	47527	15784
144	119468	43605	18960	104909	43312	15396
160	83380	27252	14028	120678	47958	18178
176	89215	29628	14890	140251	62162	19517
192	116625	48366	17057	147054	63618	20855
208	140217	47643	23135	146794	70110	19167
224	165909	67209	24666	210617	89119	30370
240	227641	96444	32789	208301	95986	28071
256	143303	49459	23445	201421	90844	27639

Table A.3: Summary of Sent Messages for `sumEuler`

number of PEs	baseline total	FISH	SCHEDULE	history-based total	FISH	SCHEDULE
64	120459	116493	989	77460	73490	990
80	143078	139101	991	103207	99228	992
96	193732	189751	992	149519	145537	993
112	242664	238681	993	170186	166204	994
128	305747	301761	994	154180	150190	995
144	371290	367309	994	241620	237634	995
160	431408	427425	994	182503	178511	996
176	376271	372281	995	253946	249956	996
192	496463	492477	995	311124	307132	996
208	578651	574661	995	384696	380705	996
224	691053	687063	995	317506	313511	997
240	809933	805944	995	372308	368305	997
256	750895	746901	996	429166	425163	997

Table A.4: Summary of Sent Messages for `parfibmap`

number of PEs	baseline total	FISH	SCHEDULE	history-based total	FISH	SCHEDULE
64	31006	27838	576	15259	10909	815
80	23961	22085	468	17059	12534	1001
96	38945	36397	635	23745	15052	1478
112	46335	43487	709	17789	13272	1128
128	56724	53720	749	16327	10933	1347
144	67495	64198	821	16830	10780	1510
160	80550	76999	886	23446	16491	1623
176	91545	87575	991	15977	9105	1717
192	107458	103106	1086	15457	8500	1737
208	119039	114580	1113	15618	9052	1639
224	130397	125527	1215	18013	9130	2115
240	145219	140062	1285	19966	13301	1662
256	150161	145439	1176	21175	14278	1722

Table A.5: Summary of Sent Messages for `parSEmap`

number of PEs	baseline total	FISH	SCHEDULE	history-based total	FISH	SCHEDULE
64	13538	12586	235	9063	7236	417
80	18281	17230	260	9795	7617	471
96	21633	20461	292	10210	8306	474
112	26585	25287	322	14160	11478	610
128	30972	29587	345	35639	32252	794
144	37503	35957	386	8877	6151	680
160	45146	43522	403	9049	6217	707
176	53051	51292	437	8334	5424	727
192	60847	58903	484	10578	7733	710
208	68991	66976	502	13842	10858	744
224	80961	78830	529	22343	19327	752
240	86920	84629	571	22370	19391	742
256	97747	95325	603	21669	18691	742

Appendix B

Implementation Details

This appendix section provides some details on the implemented extensions excerpted mostly verbatim from a companion technical report [22], which was created by the author to share this information prior to submission. Refer to the report for further details.

B.1 Compile and Run-Time Flags

When compiling a selection of flags can be used. The tables below summarise the most commonly used flags.

Table B.1: Commonly Used Compile Flags

compile-time flag	*effect*
`-O<N>`	turn on optimisation of level `N` (0 = off; typically $N = 2$)
`--make`	automatically find the dependencies and build the executable
`-fforce-recomp`	forces recompilation of used modules
`-glasgow-exts`	enables many language extensions (deprecated: use `-XextName`)
`-cpp`	enables the use of the C preprocessor for conditional compilation
`-rtsopts`	enables extended (but unsafe) RTS options
`-threaded`	links with the multi-threaded RTS (GHC-SMP)
`-eventlog`	enables event logging for `ThreadScope` (GHC-SMP)
`-parpvm or -parmpi`	links with the distributed RTS (GUM) and PVM or MPI
`-debug`	enables verbosity flags and extra sanity checks for debugging
`-prof -auto-all`	enables sequential profiling (may require profiling versions of libs)
`-i<path>`	specifies a non-standard path to a library
`-v`	verbose compiler output (e.g. to check locations searched for libs)
`-with-rtsopts<optstr>`	sets specified RTS options to different default values

Table B.2: Commonly Used Run-Time Flags

run-time flag	
`-H<size>`	suggested heap size (add e.g. `M` for Mbyte; not a limit)
`-A<size>`	suggested stack size
`-K<size>`	maximum stack size (limit)
`-M<size>`	maximum heap size
`-p`	generates a time profile
`-hT or -hC`	creates a heap profile (`.hp`, convert with `hp2ps` and `ps2pdf`)
`-G<N>`	number of generations for GC (typically 2)
GUM-specific flag	
`-qp<N>`	specifies number of GUM instances (usually one per core)
`-qPg`	prints summary statistics to `par_log_<PeID>`
`-qP`	generates event-based profile (`.gr` files)
`-qPc`	generates census-based profile (`.gs` files)
`-qz<I>`	enables history table with update interval `I` (experimental)
`-qy<L>`	enables spark co-location with lookahead bound `L` (experimental)
`-qW<S>`	wait `S` seconds at the start of execution (useful for attaching `gdb`)
`-qD<level>`	debugging output verbosity level (power of two, see Table B.3)
`-qF<N>`	sets maximum number of simultaneous `FISH` messages for a node
`-qf<N>` and `-qqf<N>`	to set `FISH` delay and `FISH` delay factor
`-qL<N>`	low watermark (start stealing if the number of local sparks is lower)
`-qT<N>`	maximum number of thunks in a packet
SMP-specific flag	
`-N<X>`	use `X` Capabilities (worker threads)
`-l[<flag>]`	creates `.eventlog` trace file for `ThreadScope`
`-S[<filename>]`	detailed GC statistics (to `stdout` or to a file)
`-s[<filename>]`	GC summary statistics
`-t`	one-line GC summary statistics
`-I<N>`	idle GC delay in seconds (experimental)

Table B.3: GUM's Debugging Output Options

option	effect (print debugging output for a sub-component)
`-qD1` or `-qDv`	verbose output related to parallel RTS in general
`-qD2` or `-qDc`	mpcomm; low level message handling
`-qD4` or `-qDp`	pack; packing code
`-qD8` or `-qDq`	packet; verbose packing
`-qD16` or `-qDP`	processes; process management
`-qD32` or `-qDo`	ports; port management code
`-qD64` or `-qDw`	weight; weights and distributed GC
`-qD128` or `-qDF`	fetching-related
`-qD256` or `-qDf`	fishing-related
`-qD512` or `-qDl`	tables; print internal address tables
`-qD1024` or `-qDd`	unused (reserved for GdH)
`-qD2048` or `-qDz`	paranoia; (creates huge output files)

B.2 Extending Victim Selection

Work stealing is a passive load distribution mechanism that assumes no knowledge about the system as idle PEs (thieves) initiate the process and select their victims at random. This has the potential to scale and has been shown to perform well on tightly-coupled shared-memory multiprocessors for well-formed workloads [38].

However, the execution starts with a single PE generating the initial sparks and in some cases (e.g. when using `parMap`) most of the parallelism will be generated early during the execution by a small set of PEs. Since idle PEs would attempt to randomly steal work, they will generate many unsuccessful stealing requests (`FISH` messages). This situation can potentially be improved by sharing and using information about the locations of past stealing successes to choose victims less randomly, increasing the likelihood of choosing PEs that have useful work to donate.

First, we inspect the implementation of work stealing beginning with the main scheduler loop that runs on each PE (see `Schedule.c`). At some point, if no threads are runnable, a local spark will be picked up and turned into a thread if available. Failing that, a `FISH` message will be sent to a PE chosen using the `choosePE()` function defined in `HLComms.c`. This is the main function to be altered to use the available history about the location of past stealing successes when choosing a PE to steal from. Below we summarise the necessary changes to extend the baseline work stealing mechanism within the RTS (see `/rts/parallel` subdirectory).

- Introduce a new data structure to hold per-PEs stealing success and failure information within each RTS instance.

- Add a new RTS option that turns on the new mechanism and takes as argument an interval that determines when the stored information can be considered out of date so it can be discarded from the history. Add the necessary initialisation and cleanup code to `RtsStartup.c` and `ParInit.c`.

- Update the code processing the incoming work request, i.e. `FISH`, messages (in the `processFish()` function in `HLComms.c`); if sparks are available, one is sent to the thief; if no work is available, the message is forwarded to another

PE, unless it has expired, in which case it is returned to the original sender. Moreover, if an own expired `FISH` message arrives, a new work request is sent to some other PE.

- Similarly, modify the code that processes the successful response to the stealing attempts, i.e. a `SCHEDULE` message with at least one spark that can be converted into a new thread (see `processSchedule()`), to update the history information.

- The packet format is extended to include the additional history information (see `sendSchedule()` in `HLComms.c` and `sendOpNV()` in `LLComms.c`), ensuring that the offsets used for packing and unpacking are properly updated. The easiest way is to extend the header but conceptually the data rather belongs into the payload. Note that endianness is important as ultimately a packet in a buffer is represented by a sequence of bytes.

- Update the `choosePE()` function to use the available history information for a less random victim selection (if the RTS option is specified).

- Add code to periodically remove stale information from the history to avoid poor choices (e.g. when processing new incoming messages).

- Enhance census-based profiling to emit history coverage (how many of the stored values are not stale for how many PE ids out of total number of PEs). This can help evaluate the appropriate choice of the invalidation interval for a given application on a specific target platform.

This is a high-level overview of the changes and the difficulty is often in correctly implementing a mechanism in detail.

B.3 Extending Spark Selection

Parallelism in GpH is exposed using `par` that is based on a primitive operation (PrimOp) `par#` from the `GHC.Conc` module[1], whereas `pseq` specifies evaluation order. To implement spark colocation we add a different version of `par` to the language by adding support for suitable primitive operations to the RTS. The new primitive, `parEnc#`, takes extra two arguments that carries some information about spark's ancestry: it includes a symbol and a base for the encoding. At run-time encoding of the parent thread is extended with a symbol from the new primitive that allows tagging of the spark with a encoding made from multiple symbols. This allows to dynamically pinpoint the location of a spark within the overall computational structure. The following are the steps necessary to add `parEnc` to the language.

1. PrimOp definition is added to `compiler/prelude/primops.txt.pp` including the PrimOp signature `primop ParInformedOp "ParEnc#" GenPrimOp` followed by the type, e.g. `a` → `b` → `Int#` → `Int#` → `b`, followed by `with` and few property specifiers like `has_side_effects = True` and `out_of_line = True`.

2. An RTS function is created that will be called by the PrimOp and perform the actual work (e.g. in `Spark.c`). It records the ancestry information for a new spark based on the parent thread's encoding and the information provided via `parEnc`. Let's call the function `sparkParEnc()`.

3. We update `inludes/stg/MiscClosures.h` with `RTS_FUN(stg_parInformedzh);` where `zh` stands for `#` in the required *z-encoding* (see GHC Wiki for more information).

4. A new symbol for `ghci` can be added to `rts/Linker.c` by extending the list of symbols with `SymI_hasProto(stg_parInformedzh)` at the end.

5. The function in `rts/PrimOps.cmm` is implemented using the conventions and features of the `Cmm` language. For instance, first eight arguments are passed in registers named `R1-R8` and can be accessed as illustrated in Listing B.1.

[1] which can be found in `libraries/base/GHC/Conc.lhs`

```
stg_parEnczh {
  W_ x;
  W_ y;
  W_ symbol;
  W_ base;
  MAYBE_GC(R1_PTR, stg_parEnczh);
  MAYBE_GC(R2_PTR, stg_parEnczh);
  x      = R1;
  y      = R2;
  symbol = R3;
  base   = R4;
  // call RTS function to enqueue a spark based on the info
  foreign "C" sparkParEnc(x "ptr", symbol, base);
  RET_P(y);
}

```

Listing B.1: Cmm Implementation of the New Primitive

After re-compiling both the compiler and the RTS (`$ make clean && make` in the `compiler` and `rts` sub-directories) along with the benchmark programs, we can proceed to test and compare the new implementation to the baseline case. Note that this discussion is based on using `GHC 6.12.3` and related `Cmm` (a variant of the `C--` portable assembly language [202]) which have meanwhile evolved further, so that particular syntax, conventions, and source file references may be different from what you will find with the most recent GHC version. We have only used and discussed the *out-of-line* PrimOps, which require minimal changes to the compiler, whilst potentially more efficient *inline* PrimOps require some changes to the code generator and are discussed on the GHC Wiki.

B.4 Extending the Profiling Component

We extend the existing profiling component to record a break-down of message counts as well as to record granularity information. In particular, thread run time is recorded in milliseconds (instead of cycles as used in GranSim-based profiling).

B.4.1 Enriching Cumulative Statistics with Detailed Message Counts

The profiling module is called `ParTicky` (see `rts/parallel/ParTicky.c` and `.h`), but the necessary changes are spread across multiple files. For instance, `-qPg` enables summary statistics, which we wish to enrich by adding information on the number of sent and received messages for different message types (e.g. `FISH` (work requests), `ACK`, `RESUME` and `FETCH`, `SCHEDULE`).

- Starting from the `rts/RtsFlags.c`, we find that the variable associated with the `-qPg` flag is `RtsFlags.ParFlags.ParStats.Global`, so we can search for its occurences to find definition and usage sites in the code and extend it to update the new counters. Note that `RtsFlags.ParFlags.ParStats.Suppressed` flag needs to be turned off and is thus included in the conditional test.

- At usage sites we find the test whether the flag is set wrapped into a preprocessor constant `PAR_TICKY` for conditional compilation, as shown in Listing B.2. It allows to re-compile the RTS to exclude this type of profiling from the code so that the overhead is not incurred if summary statistics are not required. Additionally, in the multi-threaded RTS a mutex has to be used to protect counter update. We can add similar code in other places. For instance, if we want to add some messaging-related code we would declare and initialise new counters and then use them to count messages by extending relevant parts of the `rts/HLComms.c` module. For example, an obvious place for counting sent `FISH` messages is inside the `sendFish()` function.

- Once we have added new counters (e.g. to the `globalParStats` structure) and

appropriately update them, we can print out the formatted results, similar to other results printed in `rts/parallel/ParTicky.c`.

```
#if defined(PAR_TICKY) && defined(PARALLEL_RTS)
  if ( RtsFlags.ParFlags.ParStats.Global &&
       !RtsFlags.ParFlags.ParStats.Suppressed) {
    // ...   // incrementing relevant counters
  }
#endif
```

Listing B.2: Conditional Compilation Using C Pre-Processor

Once the set of changes is complete, re-compile the RTS (`$ make clean && make` inside the `rts` sub-directory) and re-link the application with the modified RTS. Test the implementation by running some test programs with summary profiling turned on (`-qPg`) and examine the generated output.

```
// ...
 111 messages transmitted  (105 fish, \\
     2 fetch, 2 resume, 1 schedule, 1 ack)
  55 messages received     (52 fish (6 own dead), \\
     1 fetch, 1 resume (1 without GAs), 1 schedule, 0 ack)
  56 messages sent         (53 fish (7 own, 45 fwd, 1 dead), \\
     1 fetch (0 fwd), 1 resume (1 without GAs), 0 schedule, 1 ack)
// ...
```

Listing B.3: Additional Messaging Statistics (Part of the Profiling Output)

Listing B.3 presents an excerpt from the extended summary statistics found in the `par_log_*` files which also contain further cumulative statistics such as the number of sparks created and pruned as well as global address table residency for each PE. We can use a scripting language such as `perl` to parse the output files and extract the values we wish to analyse.

B.4.2 Per-Thread Granularity Profiles

As seen above, granularity profiles may provide additional insight into the computational structure of the application. Here we will look at how per-lightweight-thread granularity profiling can be added to event-based profiling (`-qP`). It is instructive to first browse the existing profiling code to understand how the related data structures are defined, initialised and used. Additionally, examining the output `.gr` files gives an idea of the output format and of necessary post-processing for data analysis and visualisation. Notably, we find the `globalParStats` structure, of type `GlobalParStats` defined in `rts/parallel/ParallelRts.h`, which could include additional information.

In GUM, each light-weight thread is implemented by a Thread State Object (TSO) that contains a pointer to the current Capability, a stack and some other book-keeping information such as the unique thread id and state (e.g. runnable or blocked). A TSO is allocated on the heap so it can be automatically garbage-collected once no longer needed after the thread terminates. We opt for avoiding to change the TSO itself as this would require non-trivial modifications to the compiler, the code generator as well as to the garbage collector, because TSO layout is crucial for efficient execution and garbage collection. Instead, we define a separate data structure to hold the information regarding execution, fetching and blocking times for each TSO (let's call it `TSOParInfo`; we also add functions for allocating and disposing the structure).

We can use a hash table to map from a thread id to the corresponding info structure. The hash table implementation is provided in `rts/Hash.c` and the API is defined in `rts/Hash.h`, which needs to be included in the files that use the hash table. The hash table can be created at RTS startup time if profiling is on and remains empty until new threads are created and info records are added to the table for each thread (e.g. in `createThread()` in `Thread.c`). When threads run or block (see `Schedule.c`, e.g. `case ThreadRunGHC`), the cumulative timers and counters inside the info structure are updated accordingly to reflect the events (`msTime()` function is used to obtain current time).

```
#if defined(PARALLEL_RTS) && defined(PAR_TICKY)
if ((tpi = lookupHashTable(globalParStats.parInfoTable,
                           cap->r.rCurrentTSO->id)) == NULL) {
    par_info = allocTSOParInfo();
    if (par_info != NULL) {
        insertHashTable(globalParStats.parInfoTable,
                        cap->r.rCurrentTSO->id, par_info);
        tpi = par_info;
    }
}

if (tpi != NULL) {
    tpi->last_time_stamp = msTime();
}
#endif

// ... run the thread using StgRun()

#if defined(PARALLEL_RTS) && defined(PAR_TICKY)
tpi = lookupHashTable(globalParStats.parInfoTable, cap->r.
   rCurrentTSO->id);
if (tpi != NULL) {     // update RT
    tpi->exectime += (msTime() - tpi->last_time_stamp);
}
#endif
```

Listing B.4: Updating Granularity Info for a TSO

Example code in Listing B.4 illustrates how the the granularity counter is updated for a given thread. When a thread terminates (see `Schedule.c`, e.g. `case ThreadFinished`), an event description is written to the event file (we model our printing function on `DumpRawGranEvent()`), before the info structure is removed from the hash table and deallocated explicitly, whilst the TSO is automatically garbage-collected.

A similar approach applies to extending the census-based profiling (`-qPc`), which is performed at particularly disruptive points in the execution such as garbage col-

lection to amortise the overhead and tends to sample the RTS state less frequently which may result in lower accuracy. On the other hand, the generated `.gs` files are usually smaller compared to the files generated using event-based profiling.

Monitoring refers to online profiling where the results are used at run time. This allows the RTS to react more flexibly to detected events. For instance, RTS can distinguish parallelism generators from workers based on the running average of the spark pool size and switch to active load distribution if deemed beneficial for some periods of time.